UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Jure Polutnik

Skalabilna oblačna storitev za analizo šahovskih pozicij

DIPLOMSKO DELO

UNIVERZITETNI PROGRAM RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: doc. dr. Boštjan Slivnik

Ljubljana, 2015

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Zasnujte, izdelajte in opišite skalabilno oblačno rešitev za analizo šahovskih pozicij. Natančno preglejte obstoječe analizatorje šahovskih pozicij, izberite najustreznejšega in ga uporabite kot jedro svoje oblačne storitve. Oblačna storitev naj omogoča horizontalno skaliranje in naj bo ustrezno testirana za različna bremena.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Podpisani Jure Polutnik sem avtor diplomskega dela z naslovom

Skalabilna oblačna storitev za analizo šahovskih pozicij

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Boštjana Slivnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 18. avgusta 2015 Podpis avtorja:

Zahvaljujem se mentorju, doc. dr. Boštjanu Slivniku, za strokovno svetovanje, usmerjanje in odzivnost pri nastajanju diplomskega dela.

Iskrena hvala moji čudoviti zaročenki Maši za vse prodorne ideje, neučakanost in opazke, ki so mi pomagale pri končanju tega dela.

Hvala moji mali miški Evi, ki mi je krajšala že tako kratke noči, a me je še vedno dan za dnem razveseljevala in polnila z energijo.

Zahvaljujem se mami Blanki in očetu Branetu, ki sta verjela vame in me na končanje tega dela opominjala ob vsakem primernem trenutku.

Posebna zahvala pa gre tudi mojemu prijatelju Noxu, ki je skupaj z menoj prebedel večere in mi ob teh težkih časih dajal vedeti, da nisem sam.

Moji dragi Evici.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
SAN	Standard Algebraic notation	Standardni algebrični zapis šahovskih potez
FEN	Forszth-Edwards notation	Forszth-Edwards zapis šahovskih pozicij
PGN	Portable game notation	Prenosljiv zapis šahovskih partij
UCI	Universal chess interface	Univerzalni šahovski vmesnik
CECP	Chess engine communication protocol	Protokol za komunikacijo s šahovskim strojem
FIDE	World Chess Federation (fr. Fédération Internationale des Échecs)	Svetovna šahovska organizacija
CEGT	Chess Engines Grand Tournament	Veliki turnir šahovskih strojev (seznam moči)
CCRL	Computer Chess Rating Lists	Seznam moči računalniškega šaha (šahovskih strojev)
GPL	GNU General Public License	Licenca za prosto programje
SaaS	Software as a Service	Programska oprema kot storitev
PaaS	Platform as a Service	Platforma kot storitev
IaaS	Infrastructure as a Service	Infrastruktura kot storitev
XaaS	Anything as a Service	Karkoli kot storitev
SLA	Service-level agreement	Sporazum o ravni storitve
NIST	National Institute of Standards and Technology	Narodni urad za standarde in tehnologijo (ZDA)
AWS	Amazon Web Services	Amazonove spletne storitve

EC2	Amazon Elastic Computer Cloud	Amazonova storitev razširljive računalniške infrastrukture
GCP	Google compute platform	Googlova storitev razširljive računalniške infrastrukture
GUI	Graphical User Interface	Grafični uporabniški vmesnik
API	Application Programming Interface	Programski vmesnik
IP	Internet Protocol	Internetni protkol
MVC	Model-view-controller	Model-pogled-krmilnik
JVM	Java virtual machine	Javanski virtualni stroj
AJAX	Asyncrhonous JavaScript and XML	Asihroni JavaScript in XML
CSS	Cascading Style Sheets	Prekrivne predloge
DOM	Document Object Model	Objektni model dokumenta
HTML	HyperText Markup Language	Označevalni jezik za oblikovanje večpredstavnostnih dokumentov
JSON	JavaScript Object Notation	JavaScript standard za zapis objektov
XML	Extensible Markup Language	Razširljiv označevalni jezik
REST	Representational State Transfer	Prenos predstavitvenega stanja
SSH	Secure Shell	Varna ukazna lupina

Povzetek

V diplomskem delu je opisan razvoj skalabilne oblačne storitve za analizo šahovskih pozicij, poimenovane ChessCloud. V ta namen je bila razvita skalabilna analitična gurča, ki analize zagotovi z uporabo trenutno najmočnejšega šahovskega stroja (Stockfish). Kot predstavitveni nivo je bila razvita odzivna in enostranska spletna aplikacija, ki omogoča pregled in analizo šahovskih partij ter igranje s šahovskim strojem na uporabniku prijazen način. Delo zajema tudi razvoj rešitve za avtomatsko pripravo oblačne infrastrukture, nameščanje storitve ter njeno upravljanje s pomočjo zmoljivih ukazov v ukazni lupini. Vsako storitev je potrebno po namestitvi tudi nadzorovati, zato delo opiše uporabljene sisteme za nadzor. Poleg tega pa je potrebno storitev tudi testirati, zato so bili pripravljeni različni scenariji, ki pokažejo, da storitev deluje stabilno tudi pri povečanih obremenitvah. Ker delo zajema cel prenos aplikacije v oblak, so lahko opisani postopki, ogrodje storitve in uporabljene tehnologije v pomoč tudi pri prenosu ostalih aplikacij oziroma orodij v oblak.

Ključne besede: računalništvo v oblaku, računalniški šah, migracija šahovskega stroja v oblak, Stockfish, Akka, OpenStack, AWS

Abstract

This thesis describes the development of scalable cloud service for analysing chess positions, named ChessCloud. For this purpose, a scalable analytic cluster has been developed that provides chess position analysis by using the state of the art chess engine implementation 'Stockfish'. As for the presentation part, a responsive single-page web application has been developed, which provides user-friendly components for analysing chess games, as well as playing a game against the chess engine. The thesis also describes the development of a solution for the automated infrastructure initialization and services deployment along with the service management through a powerful set of shell commands. Each system component is required to be monitored after the deployment, therefore the work describes the solution used for this purpose. In addition, different testing scenarios have been prepared to show that the service is stable and working correctly even with increased loads. This work covers the complete procedure to migrate an existing application to the cloud, including the technologies that make the transition smoother. Used techniques and approaches can be therefore also used as a guidance in the migration of other legacy applications into the cloud.

Keywords: cloud computing, computer chess, chess engine migration, Stockfish, Akka, OpenStack, AWS

Poglavje 1 Uvod

Namen diplomske naloge je razvoj skalabilnega oblačnega servisa za analiziranje šahovskih partij. V času, ko se večino storitev seli v oblak, vidim potrebo, da se podobno naredi tudi za storitve, ki jih uporablja številčna šahovska skupnost in za katere oblačna alternativa še ne obstaja. Trenutno se v ta namen uporabljajo namenske namizne aplikacije, povezane s šahovskimi stroji (angl. chess engines), ki uporabljajo lokalne vire in so neodvisne od spletne povezave. To seveda ima svoje prednosti, vendar menim, da trenutni ekosistem in potreba namenskih naprav, stalno povezanih v splet (npr. mobilne naprave, tablice, pametne ure), narekuje, da se tudi procesorsko zahtevni procesi prestavijo v oblak in tako zagotovijo večjo zmogljivost ter razbremenijo lokalne vire, ki so po naravi manj zmogljivi in bi morali biti namenjeni boljši uporabniški izkušnji ter obdelovanju podatkov lokalne narave (npr. lokacijske storitve). Kot primer lahko navedem Adobe Creative Cloud, s pomočjo katerega je Adobe najzahtevnejše grafične obdelave prestavil v oblak in s tem omogočil razvoj mobilnih aplikacij, ki so, kljub manjši razpoložljivosti sistemskih sredstev, sposobne podobnih operacij kot namizne aplikacije.

Cilj diplomskega dela je omogočiti zmogljivo analizo šahovskih pozicij na oblačni infrastrukturi in s tem ponuditi dodano vrednost rešitvam, ki so trenutno na voljo na mobilnih napravah in spletu. Zaradi zmogljivih tipov navideznih naprav (angl. virtual machine), ki so na voljo pri oblačnih ponudnikih, se lahko rešitev postavi ob bok tudi namenskim namiznim aplikacijam in tako odpravi potrebo po nadgrajevanju računalnikov zavoljo hitrejše in boljše analize.

Diplomska naloga zajema:

- Razvoj skalabilne oblačne rešitve s sodobnimi oblačnimi tehnologijami.
- Postavljanje in upravljanje rešitve v oblak z uporabo orodij, ki omogočajo avtomatiziranje postpokov: inicializacija infrastrukture, oskrbovanje (provisioning), postavitev (deployment).
- Horizontalno skaliranje storitve oziroma gruče.
- Nadzor sistema na nivoju infrastrukture kot tudi same storitve.
- Obremenitveno testiranje (angl. load testing) storitve, za preverjanje pravilnosti izvedbe in postavitve.

Poglavje 2 Računalniški šah

Problem, ki ga rešuje to delo, je razvoj gruče, namenjene analiziranju šahovskih pozicij. To poglavje navaja in opiše osnovne gradnike računalniškega šaha, ki jih je potrebno upoštevati pri razvoju. To so:

- grafična okolja, namenjena igranju in analizi šahovskih partij,
- šahovski stroji ter vmesniki za komunikcijo z njimi,
- šahovski zapisi oziroma notacije, ki se pri tem uporabljajo,
- osnove šahovskega stroja in uporabljeni algoritmi.

2.1 Grafična okolja

Obstaja mnogo šahovskih grafičnih okolij (odprtokodnih in komercialnih), katerih skupna točka je, da podpirajo uporabo različnih šahovskih strojev. To omogočajo standardizirani šahovski vmesniki za komunikacijo s strojem ter šahovski zapisi, ki določajo, na kakšen način se podatki med grafičnim okoljem in strojem izmenjujejo.

Šahovski stroj in njegov uporabniški vmesnik se lahko interpretira kot Model-View-Controler (MVC) arhitekturni vzorec [2], ki kot ime narekuje, razdeli prikaz (View) od stroja (Controller) s pomočjo domensko specifične reprezentacije (Model). To pomeni, da je funkcionalnost šahovskega stroja (iskanje naslednje poteze, upravljanje s časom in administracija šahovske igre) izolirana od vnosa in predstavitve.

Popularna odprtokodna okolja so Xboard, Arena ter WinBoard, med komercialnimi pa so najuspešnejša Fritz, Chessmaster ter Shredder. Slika prikazuje grafično okolje Fritz verzije 12, ki poleg množice funkcionalnosti vsebuje tri grafične kopmonent, ki jih srečamo v vsakem grafičnem okolju namenjenemu analizi šahovskih partij; komponenta za prikaz igre (šahovnica), komponenta za prikaz zapisa igre in komponenta za prikaz evalvacije trenutne pozicije.

Slika 1: Grafično okolje Fritz verzije 12

2.2 Šahovski zapisi

Skozi zgodovino se je pojavilo mnogo šahovskih zapisov, ki so služili za zapisovanje potez šahovski partiji ali za opisovanje šahovske pozicije. Prvi zapisi so bili zelo opisni, kjer so premike figur opisovali v celem stavku, skozi čas pa so se zaradi praktičnosti krajšali. Danes so v uporabi samo algebrični zapis in njegove izpeljanke, ki ga je leta 1737 uvedel Philip Stamma [1]. V nadaljevanju sta predstavljena standarden algebrični zapis in Smithova notacija, kjer je prvi splošno uporabljen v šahovski skupnosti, drugi pa predvsem v šahovskem programju.

Poleg samega zapisa premikov pa obstajajo še zapisi šahovskih partij in pozicij. Prva skupina opisuje potek partije od začetne pozicije, medtem ko je druga namenjena opisovanju poljubnih šahovskih pozicij. V nadaljevanju sta opisana najbolj pogosta zapisa iz vsake skupine (PGN in FEN) [3], ki jih srečamo v računalniškem šahu.

2.2.1 Standardni algebrični zapis

Dandanes je v spošni uporabi standardni algebrični zapis (angl. Standard Algebraric Notation - SAN), ki ga je standardizirala Svetovna šahovska organizacija (FIDE) [4]. Šahovska polja poimenuje s parom črke in števila, kjer črke (od *a* do *h*) pomenijo linije, števila (od *1* do *8*) pa vrste. Vsaka figura je označena s črko (*P - pawn = kmet, N - knight = konj, B - bishop =*

lovec, R - rook = trdnjava, Q - queen = kraljica in K - king = kralj), ki se jo doda na začetku zapisa, razen pri kmetu, kjer se oznaka zaradi krajšega zapisa lahko izpušča. Poteza se opiše tako, da se navede, na katero polje je bila določena figura premaknjena (*Nf3* - premik konja na polje *f3*). V primeru dvoumnega zapisa (npr. obe trdnjavi lahko zasedeta isto polje), se doda še dodatna informacija, ki pove polje figure pred premikom (Nbf3 - premik konja iz kolone b na pozicijo f3). Spodaj je naveden primer takšnega zapisa:

```
1. e4 e5 2. Nf3 d6 3. d4 cxd4 4. Nxd4 Nf6 5. Nc3
```

Izsek 1: Odprta Siciljanska otvoritev v standardnem algebričnem zapisu

Slika 2 : Odprta Siciljanska otvoritev

2.2.2 Smithova notacija

Smithova notacija [1] je direktna (angl. straight-forward) šahovska notacija, kjer je vsaka poteza opisana nedvoumno; ni potreben celoten potek igre, kot pri zapisu SAN, da jo je možno razbrati.

<from><to>[<capture>][<promotion>]

Sestavljena je iz polja, od koder je bila figura prestavljena <from>, s poljem, kamor je bila premaknjena <to>, in z dvema dodatnima črkama, ki sta v uporabi za neosnovne poteze <capture> (jemanja, rokade in en-passant) ter za promoviranje <promotion>. Čeprav zapis

ni kompakten kot SAN in je tudi težje berljiv, je notacija zaradi nedvoumnosti in sledljivosti (naprej in nazaj) v računalniškem šahu široko uporabljena. Spodaj je naveden primer takšnega zapisa:

```
1.e2e4 c7c5 2. g1f3 d7d6 3. d2d4 c5d4p 4. f3d4 g8f6 5. b1c3
```

Izsek 2: Odprta Siciljanska otvoritev v Smithovi notaciji

2.2.3 Zapis PGN

Zapis PGN (Portable Game Notation) [3] omogoča zapis šahovskih partij in je široko uporabljen v šahovskih grafičnih okoljih, namenjen pa je za shranjevaje in nalaganje partij. Poteze so zapisane v standardnem algebričnem zapisu, predpostavlja pa se, da zapis vsebuje celotni potek igre. Zapis je sestavljen iz glave, ki vsebuje atribute, ki opisujejo partijo (lokacija, igralci, čas, itd.), ter iz telesa, ki predstavlja zaporedje potez v partiji v standardnem algebričnem zapisu. Dodani so lahko komentarji, ki so obdani z zavitimi oklepaji. Standardni nabor atributov v glavi vsebuje:

- *Event* **:** ime šahovskega turnirja ali dvoboja.
- *Site* **:** lokacija, kjer je potekal dogodek.
- *Date* **:** čas, kdaj je potekal dogodek.
- *White* **:** ime in priimek igralca z belimi figurami.
- *Black* **:** ime in priimek igralca s črnimi figurami.
- *Result* **:** razultat šahovske partije.

```
[Event "Tal - Larsen Candidates Semifinal"]
[Site "Bled YUG"]
[Date "1965.08.08"]
[White "Mikhail Tal"]
[Black "Bent Larsen"]
[Result "1-0"]

1.e4 c5 2.Nf3 Nc6 3.d4 cxd4 {Odprta Siciljanska igra} 4.Nxd4 e6 5.Nc3
d6 6.Be3 Nf6 7.f4 Be7 8.Qf3 O-O 9.O-O-O Qc7 10.Ndb5 Qb8 11.g4 a6
12.Nd4 Nxd4 13.Bxd4 b5 14.g5 Nd7 15.Bd3 b4 16.Nd5 exd5 17.exd5 f5
18.Rde1 Rf7 19.h4 Bb7 20.Bxf5 Rxf5 21.Rxe7 Ne5 22.Qe4 Qf8 23.fxe5
Rf4 24.Qe3 Rf3 25.Qe2 Qxe7 26.Qxf3 dxe5 27.Re1 Rd8 28.Rxe5 Qd6
29.Qf4 Rf8 30.Qe4 b3 31.axb3 Rf1+ 32.Kd2 Qb4+ 33.c3 Qd6 34.Bc5
Qxc5 35.Re8+ Rf8 36.Qe6+ Kh8 37.Qf7 1-0
```

Izsek 3: Primer zapisa šahovske partije v PGN zapisu

2.2.4 Zapis FEN

Zapis FEN (Forsyth-Edwards Notation) [3] omogoča zapis šahovskih pozicij, ki je široko uporabljena v računalniškem šahu, namenjen pa je shranjevanju in nalaganju šahovskih pozicij in se uporablja predvsem v problemskem šahu. Sicer pa je zapis široko podprt tudi v šahovskih strojih, s katerimi preko vmesnika sporočimo pozicijo, ki jo želimo analizirati. Kombiniramo ga lahko tudi s PGN zapisom, katermu lahko pripnemo FEN atribut in s tem določimo začetno pozicijo; uporabno predvsem pri izvedenkah šaha, ki se ne začnejo s standardno postavitvijo (npr. Ficsher Random Chess).

Zapis FEN je sestavljen iz šestih polj, ločenih s presledkom, ki natančno določajo pozicijo, dovoljene poteze in stran, ki je na potezi.

1. Polje navaja postavitev figur (iz prespektive belega igralca) po vrsticah, ki so razdeljene s poševnico. Vsaka vrsta opisuje figure po stolpcih, kjer se za bele figure uporabljajo velike (KQRBNP), za črne pa male črke (kqrbnp). Prazna polja so označena s številko, ki pove, koliko jih je skupaj.
2. Polje navaja stran, ki je na potezi (w - beli, b - črni).

3. Polje navaja rokade, ki so na voljo. ‘K’ ali ‘Q’ beli lahko rokira na kraljevo in ali na damino stran. Za črnega se uporabijo male črke. ‘KQkq’ tako pomeni, da so mogoče še vse rokade, v nasprotnem primeru pa se navede z ‘-’.
4. Polje navaja možnost šahovske poteze en-pasant, kar se označi z oznako kolone, ki enoznačno označuje kmeta, ki ga je možno jemati. V primeru, da en-pasant ni mogoč, se polje označi z ‘-’.
5. Polje navaja število polpotez od zadnjega jemanja ali premika kmeta (uporabi se za pravilo 50 potez, kjer lahko igralca zahtevata remi).
6. Polje navaja število odigranih potez (štetje se začne z 1 in se poveča, ko črni naredi potezo).

Primer FEN formata za začetno pozicijo in pozicjo po prvih treh potezah *Siciljanske otvoritve*.

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
```

Izsek 4: Začetna pozicija v zapisu FEN.

```
rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - 1 2
```

Izsek 5: Pozicija Siciljanske otvoritve v zapisu FEN.

Slika 3: Začetna pozicija siciljanske otroviteve

2.3 Šahovski vmesnik

Šahovski stroji za sporazumevanje z grafičnimi okolji uporabljajo vmesnike, preko katerih je mogoče šahovski stroj nastaviti, zagnati ter pridobiti rezultate. V širši uporabi sta dva vmesnika: **Universal Chess Interface (UCI)** [5] in **Chess Engine Communication Protocol (CECP)** [6]. Oba vmesnika sta odprta (CECP je tudi licenciran pod GNU licenco) in za komunikacijo preposto uporabljata tekstovni standardni vhod in izhod, kjer se preko vhoda zahteva določeno akcijo, preko izhoda pa stroj sporoča rezultate te akcije.

Izmed obeh šahovskih vmesnikov se dandanes bolj množično uporablja UCI protokol, ki je novejši in prijaznejši do programerjev grafičnih okolij, saj omogoča večjo nastavljivost šahovskih strojev (npr. število procesov, velikost zgoščevalnih tabel, itd.) in razširjanje njihove funkcionalnosti (*uporaba otvoritvenih knjižnic in tabel za končnice*). Sicer ima skupnost programerjev o uporabi teh vmesnikov nasprotujoča si mnenja, vendar je očitno, da je danes v širši uporabi UCI vmesnik, medtem ko CECP novejši šahovski stroji (npr. StockFish) in tudi namenske aplikacije več niti ne omogočajo.

»I simply don't like UCI. It subsumes all engine control parameters. It tells the engine when to ponder, when to search, when to stop, etc. That is contrary to my design and I have no interest in hacking Crafty to support something that is so different from the WinBoard/XBoard protocol that has been around for a long time and which works perfectly… It removes several critical engine-decisions that are best made by the engine, not the GUI. « ***Robert Hyatt (*razvijalec šahoveskega stroja *Crafty)***

Izsek 6: Komentar zoper uporabo UCI vmesnika

» The choice of UCI is based on software-design principles that are not easy to explain. It's a programmer's thing really, I don't expect engine users to understand. Let me give you a clue though: think about young WinBoard engines that you have tried; how many handled pondering ... without bugs??? Another clue might be that surely, Stefan Meyer-Kahlen *knows a lot about good programming, right? So trust him if not me, UCI is good for programmers because it leads to fewer bugs in the code ...«* ***Fabien Letouzey (razvijalec večih šahovskih strojev)***

Izsek 7: Komentar naklonjen UCI vmesniku

Leta 2008 je vmesnik CECP sicer doživel posodobitev, s katero je omogočil nastavitve, podobno kot to omogoča UCI protokol. Posodobitev je bila narejena predvsem z namenom, da omogoči razvoj prilagojevalcev za grafična okolja, ki so pred tem uporabljala samo protokol CECP. Eden takih je XBoard, ki je v izvirni različici komuniciral preko CECP z GNU Chess šahovskim strojem (prvi referenčni šahovski stroj, ki je bil odprt in licenciran z GNU licenco in zaradi katerega je bil razvit CECP, da je bila komunikacija z XBoard okoljem sploh mogoča).

2.3.1 Vmesnik UCI

Vmesnik UCI (Universal Chess Interface) [5] je odprt komunikacijski protokol, ki so ga ponudili brez plačila licenčnine. Razvit je bil za namene komercialnega šahovskega okolja in stroja *Shredder*, ki sta ga leta 2000 razvila avtorja Rudolf Huber in Stefan Meyer-Hahalen. Vmesnik ni imel širše podpore, vse dokler ga leta 2002 ni uporabilo podjetje *Chessbase*, ki razvija popularno šahovsko okolje in stroj *Fritz*. Od tu naprej je popularnost vmesnika UCI v

primerjavi s CEPC naglo rasla in ga danes podpirajo (tudi ekskluzivno) vsi najmočnejši in najpopularnejši šahovski stroji.

Po specifikaciji je protokol razdeljen na dva dela: *vhod* (GUI to Engine) ter *izhod* (Engine to GUI). Prvi del predstavljajo ukazi, ki se pošiljajo šahovskemu stroju, drugi del pa so odgovori šahovskega stroja, ki so lahko eno- ali večvrstični. V času analize se trenutno stanje nepretrgano izpisujejo na izhod.

Spodaj je tabela najpomebnejših ukazov in odgovorov. Vsa komunikacija poteka preko standardnega vhoda / izhoda in vedno poteka v smeri šahovskega stroja, kar pomeni, da sam brez ukaza ne počne nič in samo čaka na naslednji ukaz. Stroj mora ukaz sprejeti ne glede na to, v katerem stanju je trenutno (npr. v istem trenutku poteka analiza pozicije). Če le-te v določenem stanju ne pričakuje oziroma je ne pozna, jo po specifikaciji preprosto ignorira.

Ukaz	Opis ukaza	
uci	Sporočilo šahovskemu stroju, da se bo uporabljal UCI protokol. To je po navadi prvi ukaz, ki se ga pošlje po tem, ko je stroj inicializiran. Odgovor po specifikaciji vsebuje osnovne informacije (podatke) o stroju, podprte nastavitve in zaključek z ***uciok***, s čimer sporoči, da podpira UCI protokol.	
	Izhod	Opis izhoda
	id [name | author] <string>	Sporoči ime stroja in njegovega avtorja
	option name <string> type <option_type> [default <integer>] [min <integer>] [max <integer>] [var <string>]	Sporoči možne nastavitve stroja (stil igranja, število iskalnih niti in velikost pomnilnika, časovne nastavitve itd.). Vsaka nastavitev je zapisana v svoji vrstici in vsebuje vse potrebne podatke za grafična okolja, da lahko zgradijo uporabniški vmesnik, namenjen nastavitvam stroja. Podprti tipi so **check** (zastavica, ki omogoči oz. onemogoči določeno funkcionalnost)**, spin** (nastavitev določene številčne vrednosti, ki je lahko omejena)**, combo** (izbira ene izmed predefiniranih vrednosti)**, button** (enkratna akcija, ki spremeni nastavitve; npr. ponastavi vrednosti)**, string** (nastavitev preko teksta; npr. lokacija otvoritvene knjižice). Odvisno od tipa so lahko vrednosti omejene in

<table>
<tr><td></td><td></td><td>prednastavljene z uporabo default, min/max (uporaba pri spin tipu za omejitev vrednosti), var (uporaba pri combo tipu, za navedbo vseh možnih vrednosti) spremenljivk. Spodaj so trije primeri opisov nastavitev. Prvi dve nastavitvi nastavljata konfiguracijo iskanja, število niti, ki naj se uporabijo pri iskanju in velikost pomnilnika. Tretja je tipa check in nastavi stroj, da razvija vozlišča tudi, ko le-ta ni na potezi.

Primeri:
option name Threads type spin default 1 min 1 max 128
option name Hash type spin default 32 min 1 max 16384
option name Ponder type check default true</td></tr>
<tr><td></td><td>uicok</td><td>Stroj s tem obvesti grafično okolje, da je UCI podprt ter da je zaključil s pošiljan poslal vse informacije.</td></tr>
<tr><td>setoption name [value]</td><td colspan="2">Setoption je ukaz za spreminjanje nastavitev šahovskega storja. Vrednost je potrebna za vse tipe, razen za tip button, saj le-ta samo sporoči stroju, naj izvede določeno akcijo. Ime in vrednost nastavitve razlikuje med malimi in velikimi črkami ter lahko vsebuje tudi presledke. Spodaj so navedeni trije primeri nastavitev, ki nastavijo število niti na 4 (uporabno, če je na razpolago več CPU-jev), velikost pomnilnika na 8GB ter omogočijo ponder funkcionalnost.
Primeri:
setoption name Threads value 4
setoption name Hash value 8192
setoption name Ponder value true</td></tr>
<tr><td>debug [on|off]</td><td colspan="2">Vključi oziroma izključi razhroščevalni način. Če je le-ta omogočen, stroj poleg standardnih izhodov posreduje tudi dodatne razhroščevalne (debug) informacije.</td></tr>
<tr><td>isready</td><td colspan="2">Ukaz se uporabi za sinhroniziranje stroja z grafičnim okoljem, ko le-ta pošlje enega ali več ukazov, ki lahko trajajo dlje časa.</td></tr>
<tr><td></td><td>Izhod</td><td>Opis izhoda</td></tr>
<tr><td></td><td>readyok</td><td>Stroj vedno odgovori z readyok, vednar šele, ko je zinicializiran in so vse nastavitve dokončane. V</td></tr>
</table>

<table>
<tr><td></td><td></td><td>primeru, da je stroj v stanju analize, pošlje odgovor takoj in ne po končanju analize.</td></tr>
<tr><td>ucinewgame</td><td colspan="2">Obvesti stroj, da bo naslednja analiza (iskanje) iz druge igre in naj se trenutna pozicija ponastavi.</td></tr>
<tr><td>position
[fen | startpos] moves</td><td colspan="2">Nastavi interno pozicijo, ki se bo uporabil pri naslednji analizi. Razporeditev se lahko pošlje v formatu FEN ali kot zaporedje potez od začetne. Spodaj sta primera ukazov, ki nastavita trenutno pozicijo, ki jo dobimo po prvi potezi belega in črnega 1.e4 e5. Prva posreduje razporeditev v FEN formatu, durga pa v obliki zaporedja potez v Smith notaciji:
postition fen rnbqkbnr/pppp1ppp/8/4p3/4P3/8/ PPPP1PPP/RNBQKBNR w KQkq e6 0 2
position startpos e2e4 e7e5</td></tr>
<tr><td>go
[movetime | infinite | depth | nodes | mate]
[time settings]
[searchmoves...]</td><td colspan="2">Ukaz »go« stroju sporoči, naj začne s preiskovanjem (z analizo) trenutno nastavljene razporeditve (nastavljena s pomočjo »position« komande). Iskanje je lahko neomejeno (infinite) in se konča na ukaz grafičnega okolja »stop«. Mogoče ga je omejiti časovno (movetime), glede na globino iskanja (depth), glede na število razvitih vozlišč (nodes) in na iskanje mata v določenemu številu potez (mate). Dodatno se lahko analizo omeji samo na preiskovanje vnaprej določenih potez (searchmoves), ki so navedene v Smithovem zapisu. V primeru časovno omejene igre se lahko stroju sporočijo trenutni čas posameznega igralca (wtime, btime) in časovni dodatki na potezo (winc, binc, movestogo). Spodaj sta navedena primera ukazov, kako zagnati iskanje. Prvi zažene neomejeno iskanje, drugi pa časovno in glede na globino omejeno iskanje.
go inifinite
go movetime 3000 depth 18

Stroj v času iskanja sproti sporoča informacije o iskanju ter ob koncu sporoči najoptimalnejšo najdeno potezo. Spodaj so navedeni možni izhodi.</td></tr>
<tr><td></td><td>Izhod</td><td>Opis izhoda</td></tr>
<tr><td></td><td>bestmove <move> [ponder <move>]</td><td>Po končani analizi stroj sporoči najboljšo možno potezo za dano pozicijo in opcijsko še najboljši odgovor na njo »ponder«. Poteze so navedene v Smith notaciji.</td></tr>
<tr><td></td><td>info [...]</td><td>Sporoča se med analizo. Stroj lahko sporoči eno ali več različnih informacij: depth, seldepth,</td></tr>
</table>

		time, nodes, pv, multipv, score , currline, currmove, currmovenumber, hashfull, nps, tbhits, cpuload, string, refutation. Z »depth« in »seldepth« stroj sporoča, kakšno globino je dosegel pri analizi, s »score« pove trenutno ovrednotenje pozicije (ki je lahko samo zgornja ali spodnja meja ali število potez do mata), z »nodes« število razvitih vozlišč in s »time« trenutno porabljen čas. Na koncu doda še optimalno zaporedje potez »pv« za dano začetno potezo (stroj lahko nastavimo, da razvija več začetnih potez hkrati; z uporabo **MultiPV** nastavitve). info depth 20 seldepth 25 score cp 19 nodes 5180071 nps 1544445 time 3354 multipv 1 pv e2e4 e7e6 b1c3 d7d5 d2d4 f8b4 f1d3 d5e4 d3e4 g8f6 c1g5 b8d7 g1e2 h7h6 g5f6 d7f6 e1g1 f6e4 c3e4 e8g8 info nodes 3911116 time 2561
stop	Sporoči stroju, naj preneha z analiziranjem ter izpiše najboljšo najdeno potezo »bestmove«.	
quit	Sporočilo stroju, naj se zaustavi**.**	

Tabela 1: UCI vmesnik – ukazi in izhodi [5]

2.3.2 Algoritmi

Problem analize šahovskih pozicij je tipičen primerek iskalnih algoritmov v teoriji iger s popolno informacijo in neomejenim prostorom. Glavna dela šahovskega algoritma sta iskalna strategija in evluacijska funkcija. Pri prvi gre za način preiskovanja iskalnega drevesa, pri katerem se večinoma uporabljajo izpeljanke minmax iskalnega algoritma z alfa-beta razanjem[1] (PVS, NegaScout, MTD(f) ...), ki razvijajo iskalno drevo po principu prvo globina (angl. depth-first). Obstajajo še algoritmi, ki uporabljajo tudi principu prvi najboljši (angl. best-first). S pomočjo evalvacijske funkcije pa algoritem hevristično oceni dano pozicijo (vozlišče), ki se uporabi za nadaljnje preiskovanje iskalnega drevesa, in oceno (tj. možnosti za zmago) trenutne preiskovane poti. Evaluacijska funkcija navadno ovrednoti pozicijo glede na material ter strateške lastnosti pozicije (npr. dvojni, prosti in blokirani kmetje, mobilnost figur, varnost kralja …). Večino evalvacijskih funkcij je linearna kombinacija utežnih lastnosti:

[1] https://chessprogramming.wikispaces.com/Alpha-Beta

$$Eval = \sum_{i=1}^{n} Fi * Wi$$

Pri problemu analize šahovskih pozicij optimalne rešitve zaradi velikosti preiskovalnega prostora ni mogoče najti (ocena poti poda verjetnost pozitivnega oziroma negativnega izida), a je rešitev v poprečju boljša, če se razvije več vozlišč v isklanem drevesu (pri predpostavki, da strategija iskanja in evalvacijska formula ostajata enaki), kar pa je premosorazmerno s časom, procesorsko močjo in velikostjo pomnilnika. Ker pa so na mobilnih in spletnih aplikacijah strojni viri omejeni, se čas, potreben za razvijanje enakega števila vozlišč, poveča. [2]

2.3.3 Šahovski stroji

Šahovski stroj (angl. Chess engine) je program namenjen analiziranju šahovskih pozicij in predlaganju najboljših potez. Kot je navedeno v prejšnjih poglavjih, šahovksi stroj ni namenjen neposredni komunikaciji z uporabnikom, ampak preko standardiziranih vmesnikov ponuja funkcionalnost grafičnim okoljem, ki so namenjena uporabniški izkušnji. Skupnost, ki razvija šahovske stroje, je zelo velika, zaradi česar je na voljo mnogo strojev, komercialnih in zastonjskih. Pri slednjih je velika večina tudi odprtokodna.

Za določitev moči strojev obstajajo skupnosti, ki redno preverjajo nove verzije strojev v različnih formatih iger (hitropotezni, standardni itd.). Najbolj znani organizaciji sta CEGT (Chess Engines Grand Turnament) [7] in CCRL (Computer Chess Rating List) [8].

#	Ime	Elo	Licenca
1	Stockfish 6 64-bit 4CPU	3311	Odprtokodna
2	Komodo 8 64-bit 4CPU	3303	Komercialna
3	Houdini 4 64-bit 4CPU	3271	Komercialna
4	Fire 4 64-bit 4CPU	3219	Odprtokodna
5	Gull 2.8b 64-bit 4CPU	3206	Odprtokodna
6	Equinox 3.20 64-bit 4CPU	3188	Odprtokodna

7	Critter 1.6a 64-bit 4CPU	3175	Zastonjska
8	Rybka 4 64-bit 4CPU	3161	Komercialna

Tabela 2: Seznam najmočnejših šahovskih strojev

Tabela prikazuje najboljših osem strojev po 40/40 sistemu (40 minut za vsakih 40 potez) CCLR organizacije. Moč je določena s prirejenim ELO rating sistemom, ki se originalno uporablja za merjenje moči šahistov, vendar je osnova prirejena računalniškim programom. Relevantne primerjave ni, ker je iger med najboljšimi šahisti in šahovskimi stroji veliko premalo za natančno oceno, zato velja, da so trenutni šahovski stroji vsaj za red močnejši in je osnova za ELO rating postavljena med 3100-3300, kar je veliko več,kot je trenutni rating najboljšega šahista (še nihče ni presegel 2900). Sicer izmerjena ELO vrednost ni popolnoma verodostojna, ker je bila srednja vrednost izbrana glede na premoč strojev nad najboljšimi šahisti, ki so bili pripravljeni igrati (prvi računalnik, ki je premagal svetovnega šahovskega prvaka, je bil leta 1996 Deep Blue).

Iz tabele je mogoče razbrati, da je polovica najboljših šahovskih strojev odprtokodnih, med njimi tudi Stockfish verzije 6, ki je že dlje časa v samem vrhu, trenutno pa zaseda celo prvo mesto. Avtor je Norvežan Tord Romstad, ki ga je prvotno poimenoval Glaurung. Sedanje ime pa je stroj dobil, ko je razvoj nadaljeval Marco Costalba in pomeni posušeno ribo, ki je eden izmed znanih norveških izvoznih artiklov. Izbiri imena je po vsej verjetnosti botrovala Rybka, dolgo najmočnejši šahovski stroj, ki v češčini pomeni riba. Stroj podpira vmesnik UCI in je zato uporaben preko množice grafičnih okolij. Podpira do 128 procesnih jeder in 1TB velikost transpozicijskih tabel, ki jih uporabljajo za preiskovanje igralnega drevesa. Implementira napreden alfa-beta iskalni algoritem in uporablja bitno reprezentacijo šahovske plošče, ki minimizira procesorske inštrukcije ter optimizira porabo pomnilnika. Stroj je znan po veliki iskalni globini, ki jo doseže z agresivnim rezanjem iskalnega drevesa in poznim odstranjevanjem začetnih potez (angl. Late Move Reductions).

Poglavje 3 Računalništvo v oblaku

Razvita storitev ChessCloud je zasnovana na način, da omogoča izrabo lastnosti, ki jih ponuja oblak. To poglavje je namenjeno opisu lastnosti računalništva v oblaku, opisu izvedbenih modelov, ki jih največkrat srečamo, ter opisu storitvenih modelov, s katerimi oblak razdelimo v več ravni, glede na funkcionalnosti, ki jih posamezen nivo ponuja.

3.1 Lastnosti

Izraz računalništvo v oblaku izvira iz shematskega prikaza medmrežja, ki je navadno prikazan kot oblak. Izraz se je začel uporabljati v času, ko je dostopnost do medmrežja naraščala, s čemer je naraščalo tudi število storitev, dostopnih preko internetnega omrežja. S hitro prilagoditvijo pametnih naprav (telefoni, tablični računalniki in ostale nosljive naprave) se povečuje potreba po neodvisnosti naprav in dostopnosti.

Slika 4: Prikaz povezljivosti različnih naprav z oblačnimi storitvami

Namen računalništva v oblaku je uporabnikom ponuditi visoko zmogljive, zanesljive, prilagodljive in cenovno ugodne računalniške storitve, dostopne preko enostavnih spletnih vmesnikov. Glavne lastnosti računalništva v oblaku so [9] [11]:

- ***Dostopnost*:** storitve in podatki so dostopni preko internetnega omrežja; torej niso odvisni od uporabnikove lokacije.
- ***Skalabilnost*:** je zmožnost sistema, da glede na spremenjene obremenitve (npr. povečana potreba po servisu) zadosti potrebam z dodajanjem dodatnih virov ali z močnejšo strojno opremo (angl. scale-up) ali z dodajanjem dodatnih vozlišč (scale-out).
- ***Elastičnost*:** oziroma prožnost je sposobnost servisa (navadno infrasturkure) sprotnega dodeljevanja virov glede na potrebe.
- ***Agilnost*:** je lastnost, ki uporabnikom omogoča hiter in cenovno ugoden dostop do oblačnih servisov - tehnologije, virov ...
- ***Zanesljivost*:** je lastnost, ki zagotavlja redundanco strežnikov in podatkov z ostalimi pripadajočimi varnostnimi mehanizmi.
- ***Varnost:*** osnovno zagotavljajo ponudniki storitev, ki večinoma integrirajo (vključujejo) naprednejše in celovitejše sisteme za zagotavljanje varnosti, ki si jih manjša podjetja težje privoščijo.
- ***Boljša izraba sredstev (angl. utalization) in nižji stroški*:** ponuja model »plačaj po porabi« (angl. pay-as-you-go), ki omogoča vstop na trg brez investicij v strojno opremo in ostalo tehnologijo. Poleg tega se zmanjša tudi potreba po IT osebju, ki skrbi za infrastrukturo (temeljne naprave).

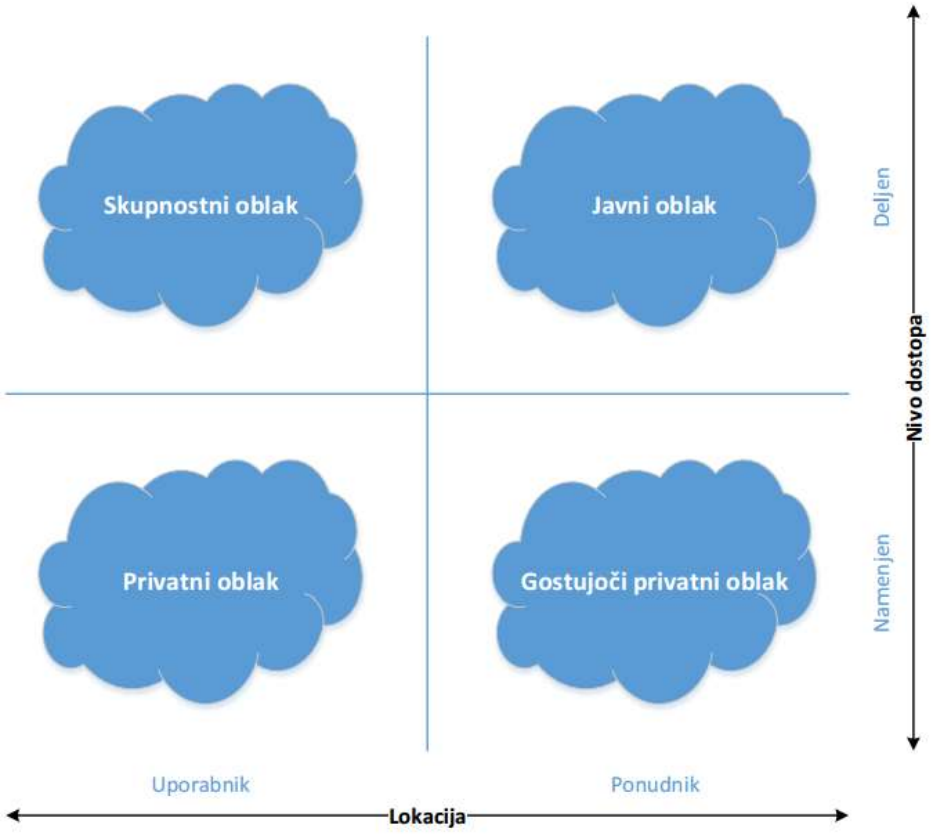
Računalništvo v oblaku ima kot vsaka tehnologija svoje prednosti in slabosti, ki se jih je potrebno zavedati, predno začnemo uporabljati storitve, ki jih ponuja oblak. Te so sicer odvisne tudi od izvedbenega modela, ki so opisani v naslednjem poglavju, tu pa je namen predstavitve slabosti javnega izvedbenega modela, ki ga navadno pripisujemo pojmu oblak. Prednosti predstavljajo že opisane lastnosti. Slabosti računalništva v oblaku (za javni izvedbeni model) pa so:

- ***Odvisnost od internetne (medmrežne) povezave:*** Ker je za dostop do storitev, nameščenih v oblaku, potrebna medmrežna povezava, sotritev za uporabnika ni na voljo, v kolikor le-ta nima povezave. Tudi v primeru slabe povezave je storitev lahko zelo zmanjšana, ali celo neuporabna.
- ***Varnost podatkov*:** Kar je prednost, je lahko tudi slabost. V primeru, da pride do uspešnega napada na samega ponudnika oblaka ali oblačne storitve, so v nevarnosti vsi shranjenji podatki. Pri izbiri ponudnika je zato potrebno biti še posebno previden.

- ***Stalno odplačevanje in lastništvo***: Model »plačaj po porabi« se lahko na daljši rok izkaže tudi dražji kot stroški, povezani z vzpostavitvijo zasebne infrastrukture in njenim upravljanjem.
- **Vezava na ponudnika (angl. vendor lock-in):** Pri izbiri ponudnika se zavežemo uporabi storitev, ki jih ta ponuja. Do problema pa pride, ko hočemo svojo storitev prenesti na drugega ponudnika. Prenos je lahko zelo otežen, tako da je pri izbiri oblaka potrebno misliti tudi, kako močna je vezava in ali bo mogoče prenesti storitve, če se bo izkazala potreba.

3.2 Izvedbeni modeli

Računalništvo v oblaku ni izdelek, tehnologija ali standard, ampak je metoda za zagotovitev prilagodljivih skupnih storitev IT v širšem pomenu. S tega vidika je logično, da se za pojmom oblak skrivajo številni vidiki, ki jih je potrebno ustrezno umestiti. Zaradi lažje predstave je bilo računalništvo v oblaku že kmalu razdeljeno na dve značilni pojavni obliki, ki imata skupne določene lastnosti, vendar sta si s stališča lastništva nasprotujoči. Ti dve obliki sta javni (angl. public) in zasebni (angl. private) oblak. Izkazalo se je, da takšna delitev vsem organizacijam ne ustreza, zato se je poleg lastniške kategorije vključila tudi lokacija postavitve in nivo dostopa. Tako trenutna definicija inštituta NIST (National Institute of Standards and Technology) [9] dodaja še hibridni (angl. hybrid) oblak in oblak skupnosti (angl. community). Slika prikazuje vse izvedbene modele glede na lokacijo postavitve in nivo dostopa.



Slika 5: Izvedbeni modeli oblaka

3.2.1 Javni oblak

Javni oblak predstavlja oblačne storitve, ki jih ponuja trg. Namenjene so splošni javnosti. Ker so viri, ki jih ponuja oblak, dostopni vsem, je vprašanje glede varnosti, prava in dostopnosti toliko pomembnješe; je pa tudi področje, kamor ponudniki vlagajo največ energije. Tehnologija javnega in zasebnega oblaka je sicer podobna, vendar je obravnava varnosti (pri dostopu do aplikacij, hrambe in ostalih virov) bolj celostna pri prvem. Največji ponudniki oblačnih storitev so med drugim tudi največji igralci v IT sektorju: *Amazon Web Services (AWS)* [13], Microsoft *Azure* in *Google Cloud Platform (GCP)*.

Prednosti javnega oblaka so, da podjetjem omogoča, da storitve najamejo in se tako izognejo investicijam v lastno infrastrukturo in IT sektor. Zelo pomemben pa je tudi dostop do aplikacij in storitev, saj ta model skoraj samodejno omogoča dostop do le-teh, neodvisno od lokacije in časa.

Slabosti javnega oblaka pa je nezmoožnost popolnega nadzora nad infrastrukturo in storitvami, saj je le ta omejena s pogodbeno dogovorjenim pravilnikom (SLA – Service Level Agreement). To lahko, zaradi dolgotrajnosti reševanja nesporazumov in neupoštevanje

pogodbenih zavez, predstavlja dodatna tveganja, ki jih je potrebno upoštevati pri izbiri ponudnika. Številni pomisleki se pojavaljajo tudi glede varnosti, dostopnosti, pa tudi zmogljivosti, ki jih je potrebno natančno preveriti.

3.2.2 Zasebni oblak

Zasebni oblak je izvedbeni model, pri katerem je celotna infrastruktura v domeni ene organizacije (podjetja), nad katero ima popoln nadzor in katere storitve so namenjene izključno le-tej. Tako kot pri drugih modelih tudi zasebni oblak zagotavlja računske in hrambene strotive iz množice strojnih virov, z razliko da so pod popolnim nadzorom (specificiranje, arhitektura in upravljanje) te organizacije. Rešitev, ki omogočajo postavitev zasebnega oblaka, je mnogo. Komercialne rešitve zasebnega oblaka ponujajo med drugimi *Oracle, HP, VMware, Cisco*. Kar bolj veseli, je dejstvo, da obstaja izredno popularna odprtokodna rešitev OpenStack [14], ki ima ogromno skupnost, med katerimi so najpomembnejši *IBM*, *Intel, RedHat, HP, Rackspace, Ubuntu in SUSE*.

Eden izmed razlogov za izbiro takšnega modela je predvsem varnost (zasebnost) nad podatki, saj omogoča uveljavljanje lastnih varnostnih standardov in popolno fleksibilnost nad infrastrukturo. Izvedba navadno predvideva, da je oblak nameščen na privatno strojno opremo (angl. on-premise), dostopen pa preko zasebnih omrežnih povezav (angl. private network links), katerega viri se lahko uporabijo samo znotraj privatnega omrežja.

Prednost zasebni oblaka je, da omogoča popolno kontrolo nad infrastrukturo in storitvami. Omogoča individualne prilagoditve tako glede varnosti kot ostalih nastavitev. Tako nadomestijo slabosti javnega oblaka, vednar ima še vedno prednosti, ki jih srečamo v računalništvu v oblaku (samopotrežba virov, avtomatizacija, visoka izkoriščenost skupnih virov, virtualizacija, fleksibilnost).

Slabost zasebnega oblaka v primejavi z javnim oblakom je, da je potrebno za takšen izvebeni model najprej investirati v infrastrukturo, nato pa plačevati stroške vzdrževanja in upravljanja. Najpogosteje je potrebno imeti močno IT službo, ki skrbi za celotno rešitev. S tem se del prednosti računalništva v oblaku zmanjša, vendar je lahko rešitev na dolgi rok ugodnejša.

3.2.3 Oblak skupnosti

Oblak skupnosti (angl. Community Cloud) je model, ki omogoča deljenje virov in storitev med več organizacijami, ki imajo med seboj podobne cilje oziroma namene. Z oblakom lahko upravljajo organizacije same (angl. on-premises), ali pa je to delo prepuščeno tretji osebi (angl. off-premises). Z odpravljanjem podvajanja podobnih sistemov lahko oragnizacije (z

enakimi zahtevami in z enako bazo uporabnikov) prihranijo pri investicijah in upravljalskih stroških. Pogosto se ta tip oblaka uporablja pri povezovanju vladnih oziroma državnih sektorjev, kar pomeni, da so storitve pogosto prilagojene potrebam posameznih skupin uporabnikov (npr. davčni svetovalci, odvetniki, trgovci).

3.2.4 Hibridni oblak

Hibridni oblak je kombinacija več izvedbenih modelov (javni, zasebni, skupnostni), kjer posamezni oblaki ohranijo lastnosti in prednosti, a so povezani v enoto, ki navzven deluje kot ena. Model je precej uporabljen pri kombinaciji (sestavi) javnega in zasebnega oblaka, kjer se občutljivi podatki hranijo v zasebnem, viri za ostale nekritične storitve pa se po potrebi pridobijo v javnih oblakih.

Prednost hibridnega oblaka je, da ponuja kombinacijo fleksibilnosti javnega oblaka in zanesljivosti zasebnega oblaka. Storitve se tako lahko na elastičen in dinamičen način odzovejo na vsa stanja obremenitve, hkrati pa se ohrani nadzor na vseh bistvenih področjih (npr. hramba občutljivih podatkov, izvajanje pomembnih procesov).

Slabost hibridnega oblaka pa je, da z uporabo več oblačanih infrastruktur tudi rešitve, ki jih uporabljajo, postajajo kompleksnejše. Poleg vzdrževanja celotnega zasebnega oblaka je potrebno tudi rešitve prilagodit na način, da so sposobne izrabiti dodatne vire v javnem oblaku.

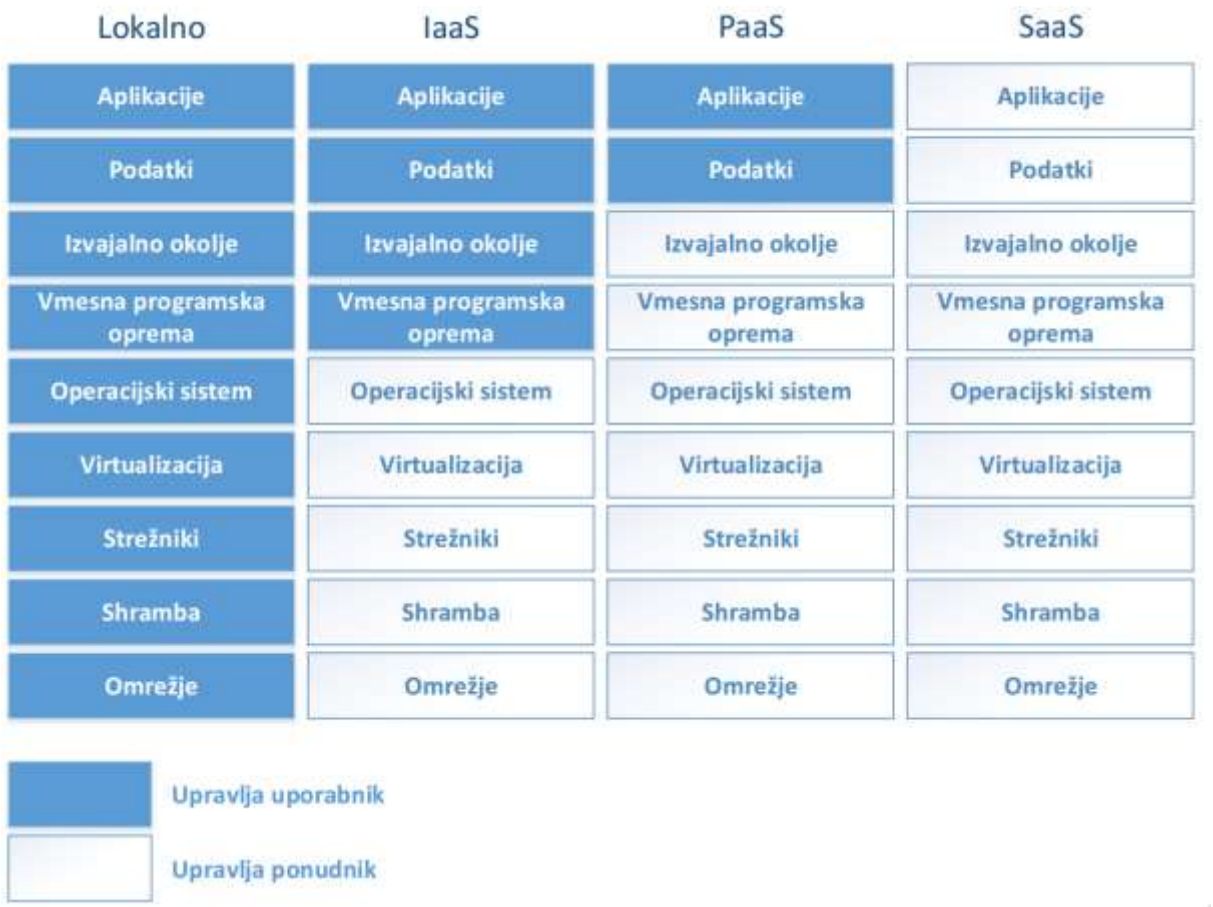
3.3 Storitveni modeli

Računalništvo v oblaku se poleg temeljnih izvedbenih oblik NIST [9] deli tudi na tri temeljne storitvene ravni (angl. service models), ki izpostavljajo in definirajo zmožnosti posameznega nivoja. Te storitvene ravni so: infrastruktura kot storitev (angl. Infrastructure as a Service - IaaS), platforma kot storitev (angl. Platform as a Service - PaaS) ter programska oprema kot storitev (angl. Software as a Service - SaaS).

Na splošno pa se ravni storitev v oblaku klasificirajo kot XaaS (angl. Anything as a service), kjer X predstavlja poljubno storitev (na primer programsko opremo, shrambo ali infrastrukturo). Poleg osnovnih storitvenih modelov poznamo še BPaaS (Business Process as a Service), DaaS (Database as a Service), IDaaS (Identitity as a Service), STaaS (Storage as a Service), itd.

Slika 6: Hierarhija storitvenih modelov

Slika ponazarja hierarhijo osnovnih storitvenih modelov oziroma ravni. Zaradi hierarhične delitve se je za vsako raven uveljavil pojem *storitvena platforma*. Splošno velja, da nižje kot smo v piramidi, večja je stopnja standardizacije, možnosti uporabe pa so bolj splošne narave. Višja kot je raven, bolj specifično je okolica prilagojena potrebam. Ker se posamezni nivoji gradijo na prejšnjih, omogoča uporabnikom storitve, da se osredotočijo na ključne funkcionalnosti svojega sistema, ne da bi jim bilo potrebno skrbeti za spodnji nivo. Uporabljajo storitve preko standardiziranih mehanizmov in protokolov. Slika podaja ilustrativno primerjavo med posameznimi nivoji in storitvami, ki so v domeni ponudnika in na drugi strani uporabnika.



Slika 7: Storitveni modeli oblaka

3.3.1 Infrastruktura kot storitev

Infrastruktura kot storitev (angl. Infrastructure as a Service - IaaS) je najbolj osnovna vrsta storitvenega modela, ki uporabniku omogoča uporabo strojnih oziroma navideznih virov preko standardnih vmesnikov, s pomočjo katerih lahko zgradi poljubno računalniško okolje. IaaS oblaki poleg standardne strojne opreme navadno ponujajo tudi hrambene vire (angl. storage resources), omrežne nastavitve, nastavitev požarnih zidov (angl. firewalls), nastavitev porazdelilnikov obremenitev (angl. load balancer) itd.

V tem storitvenem modelu ponudnik obdrži popolno kontrolo nad strojno opremo in ostalimi viri (npr. programsko opremo – angl. hypervisor), medtem ko uporabnik skrbi za nameščeno programsko opremo in nastavitve ostalih najetih virov. Uporabniki tako pridobijo cenovno ugodno razširljivost, saj jim ni potrebno vlagat v strojne vire in skrbeti za njihovo vzdrževanje.

3.3.2 Platforma kot storitev

Platforma kot storitev (angl. Platform as a Service - PaaS) je vrsta oblačne storitve, ki omogoča uporabnikom, da razvijajo spletno aplikacijo ter z njo upravljajo brez potrebe po gradnji kompleksne infrastrukture, saj je le-ta v domeni ponudnika storitve. To uporabniku zagotavlja hitrejši in cenejši razvoj aplikacije, saj se mu ni potrebno ukvarjati z administracijo sistema in namestitvijo programske opreme. To pa ima tudi slabost, saj je uporabnik omejen na storitve, ki jih platforma ponuja.

3.3.3 Programska oprema kot storitev

Programska oprema kot storitev (angl. Software as a Service - SaaS) je vrsta oblačne storitve, ki omogoča najem programske opreme oziroma spletnih aplikacij, ki tečejo na oblačni platformi. Prednost te storitve je, da uporabniku ni potrebno nameščati programske opreme, skrbeti za programske in varnostne posodobitve, kar je v domeni ponudnika. SaaS je najbolj uporabljena vrsta storitve, namenjena končnim uporabnikom in ima omejen nabor funkcionalnosti in prilagoditev.

Poglavje 4 Uporabljene tehnologije

V tem poglavju je pregled tehnologij in storitev, ki so uporabljene pri razvoju rešitve ChessCloud. Prvi del opisuje oblake, v katerih je storitev nameščena in orodja, s katerimi je namešanje olajšano in avtomatizirano. Sledijo knižice ter tehnologije, ki so uporabljene pri razvoju in se navadno uporabljajo za razvoj skalabilnih, visoko zmogljivih storitev. Na koncu pa so navedene še rešitve s katerimi je moč storitve v oblaku nadzorovat in testirat.

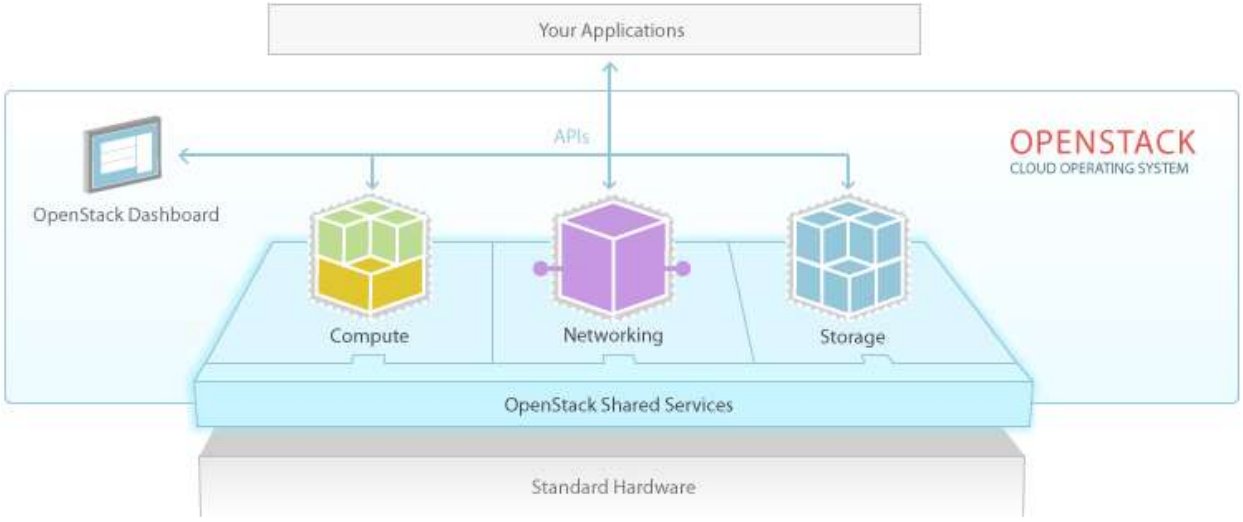
4.1 Oblak

Razvita storitev za delovanje potrebuje nadzor nad infrastrukuto, zato so primerni oblaki, ki ponujajo infrastrukturo kot storitev. V tem poglavju sta opisani rešitvi javnega (Amazon Web Services EC2) in zasebnega (OpenStack) izvedbenega modela, ki sta podprti v postavitvenih skriptah in na katerih je bila rešitev nameščena.

4.1.1 OpenStack

OpenStack [14] je odprtokodni projekt, ki zagotavlja računalniške storitve v oblaku in je namenjen postavitvi zasebnih oblakov. Projekt upravlja fundacija OpenStack, ki je bila ustanovljena leta 2012 in kateri je pridružnih veliko tehnoloških velikanov, kot so Intel, Canonical, Red Hat, Cisco, Dell, HP in IBM. Namen projekta OpenStack je zagotavljanje skalabilnega in elastičnega računalništva v oblaku v javne in zasebne namene, tudi različnih velikosti.

OpenStack ima modularno arhitekturo in je sestavljen iz množice komponent, ki skrbijo za nadzor računskih storitev, shrabmbo podatkov in nadzor nad omrežnimi viri. Poudarek je na ponujanju infrastrukture kot storitve (IaaS), razvijajo pa se tudi komponente, ki nad tem gradijo naslednji nivo storitev.



Slika 8: Visokonivojska shema OpenStack komponent

Slika prikazuje osvnovno delitev OpenStack servisov (komponent ali sestavnih delov):

- ***OpenStack Compute (Nova)*** ponuja platformo za upravljanje z navideznimi računskimi viri in je glavna komponenta okolja OpenStack, s katerim je omogočena infrastruktura kot storitev.
- ***OpenStack Networking (Neutron)*** ponuja izvedbo navideznih omrežij med virtualnimi računalniki in njihovo povezavo navzven.
- **OpenStack Storage**
  - *Block Storage (Cinder)* ponuja navidezno shrambo v obliki blokovnih polj, ki jih uporabljajo *Compute* instance navideznih računalnikov.
  - *Object Storage (Swift)* skalabilna redundantna stortiev za hrambo.
- ***OpenStack Shared Services***
  - *Image Service (Glance)* omogoča upravljanje (odkrivanje, registracijo in dostavljanje) s slikami navideznih računalnikov in diskov.
  - *Identity Service (Keystone)* ponuja centralni nadzor nad avtentikacijo in avtorizacijo (pooblastilo, dovoljenje) uporabnikov, ki ga je mogoče povezati z obstoječimi aktivnimi imeniki (npr. uporaba LDAP).
  - *Telemetry Service (Ceilometer)* je namenjen agregiranju oz. sestavljanju podatov o uporabi in zmogljivosti servisov, nameščenih v OpenStack oblak.
  - *Orchestration Service (Heat)* ponuja agilno pripravo navidezne infrastrukture z uporabo predefiniranih vzorcev oz. predlog. Preko namenskega jezika omogoča določitev računskih, hrambenih in omrežnih nastavitev kot tudi samodejnega oskrbovanja infrastrukture in nameščenih aplikacij.

- *Database Service (Trove)* je podatkovna baza, razvita z namenom popolne integracije z OpenStack funkcionalnostmi, ki uporabnikom omogoča hitro in preprosto postavitev relecijske podatkovne baze brez nepotrebne administrativne kompleksnosti.

- ***Dashboard (Horizon)*** je spletna aplikacija, ki omogoča nadzor nad oblakom, namenjena predvsem uporabnikom storitve (ne administratorjem).

4.1.2 Amazon Web Services

Amazon Web Services (AWS) [13] je oblačna platforma, ki jo sestavlja množica spletnih storitev nameščenih v enajst geografiskih regij po vsem svetu. AWS ponuja Amazon.com, ki je leta 2006 svojo infrasturkturo namenjeno spletni trgovini preoblikoval v storitev in jo ponudil javnosti. Kot prvi ponudnik je AWS postal glavni igralec v sektorju oblačnih storitev in trenutno zavezma eno tretjino celotnega trga.

Množica storitev, ki jih ponuja Amazon, je velika in se še naprej veča. Storitve so namenjene oblačni infrastrukturi in omrežju, hrambi in podatkovnim bazam, admintraciji sistemov in varnosti, nameščanju in upravljanju s sistemi, analitiki podatkov, itd. Poleg teh Amazon ponuja tudi aplikacijske in mobilne storitve, ki jih je možno vgraditi v obstoječe sisteme in okolja. Glavni sklopi z vidika infrastrukture s pripadajočimi storitvami so:

- **Računski viri (angl. Compute)**
  - EC2 je storitev, ki ponuja razširljivo platformo za upravljanje z računskimi viri v oblaku. Omogoča popolni nadzor nad navideznimi napravami (angl. virtual machine). Ponuja pa tudi storitve s pomočjo katerih je postavitev in upravljenje infrastrukture preprostejše; storitev, ki omogoča avtomatsko skalabilnost (angl. auto-scalability), elastična porezdelitev obremenitev (angl. elastic load-balancer).
  - Lambda je storitev, ki zažene naložene rutine kot odziv na zunanje dogodke ter upravlja z računskimi viri, zaradi česar uporaba storitve preprosta.
  - EC2 Container Service je storitev, ki omogoča nalaganje, upravljanje in skaliranje Docker kontejnerjev, ki so zadolženi poganjanje aplikacij in storitev.
- **Omrežje (angl. Networking)**
  - VPC je storitev, ki omogoča postavitev AWS virov v virtualno mrežo.
  - DirectConnect je storitev, ki omogoča preprosto in varno povezovanje lokalnih (angl. on-premises) storitev z Amazonovimi storitvami.

  - Route 53, je storitev, ki omogoča skalabilen in zanesljiv oblačni domenski strežnik (DNS).

- **Hramba in dostava vsebine (angl. Storage & Content delivery)**
  - S3 (Simple Storage Service) je storitev, ki ponuja varno, zanesljivo in skalabilno objektno hrambo.
  - Glacier je storitev, ki ponuja prostor namenjen arhiviranju in shranjevanju varnostnih kopij.
  - CloudFront je storitev namenjena dostavi vsebin do končnih uporabnikov. Omogoča prenos z minimalno zakasnitvijo in visoko hitrostjo prenosa podatkov.
- **Podatkovne baze (Database)**
  - RDS (Relational Database Service) je storitev, ki omogoča preprosto postavitev in upravljanje relacijske baze v oblaku. Omogoča vertikalno in horizontalno (replikacije) skalabilnost.
  - DynamoDB je oblačna rešitev NoSQL posdatkovne baze, ki omogoča visoko in predvidljivo zmogljivost.
  - ElastiCache je storitev, ki omogoča enostanvno namestitev, upravljanje in skaliranje podatkovnega predpomnilnika (angl. in-memory cache)
  - Redshift je storitev, ki omogoča hrambo velike količnine podatkov (petabyte) in analiziranje le teh z obtoječimi poslovnimi orodji.

4.2 Namestitev

Razvita sotritev za namešanje v oblak uporablja orodja namenjena avtomatizaciji tega opravila. S kombinacijo opisanih tehnologij so za namen ChessCloud storitve pripravljene preproste, toda zmogljive skripte, ki omogočajo inicializacijo infrastrukture, nameščanju storitve ter njeno upravljanje.

4.2.1 Apache LibCloud

Apache LibCloud [15] je odprtokodna Python knjižica, namenjena komunikaciji z oblačnimi storitvami preko enotenga vmesnika. Ustvarjena je bila z namenom, da razvijalcem ponudi preprosto orodje, ki deluje z najbolj popularnimi servisi, ki jih je moč najti v oblačni sferi. LibCloud podpira delno ali v celoti več kot 30 ponudnikov oblačnih storitev. Med njimi so najpomembnejši *Amazon Web Services*, *Google Cloud Platform*, *OpenStack*, *Microsoft Azure.*

Viri, ki jih je mogoče upravljati z LibCloud rešitvijo, so v grobem razdeljeni v naslednje skupine:

- ***Compute in Block Storage services*** sklop omogoča upravljanje z elastičnimi infrastrukturami in blokovnimi hrambami, ki so namenjene navideznim računalniškim storjme. Primer prvega sta *OpenStack Nova (Compute service)*, *Cinder (Block storage service)* in *Glance (Image service)* ter *Amazon EC2*, ki ponuja celotni sklop pod istim servisom.
- ***Object Storage services*** sklop je namenjen storitvam, ki ponujajo hrambo podatkov in hitro dostavaljanje podatkov do uprabnikov. Primer predstavljata *OpenStack Swift* in *Amazon S3* ter *CloudFront*.
- ***Load Balancers as a Service (LBaaS)*** omogoča upravljanje s servisi, ki ponujajo porazdelilnike obremenitev. Primer predstavljata *OpenStack Neutron/LbaaS in Amazon Elastic Load Balancer.*
- ***DNS as a Service (DNSaaS)*** omogoča upravljanje s storitvami, ki ponujajo elastične DNS storitve. Primer predstavljata storitvi *OpenStack Designate* in *Amazon Route 53.*

4.2.2 Fabric

Frabic [16] je Pyhon knjižica, namenjena nameščanju aplikacij in upravljanju administrativnih nalog, ki izvede visokonivijski vmesnik nad varno lupino (angl Secure Shell - SSH). Ponuja osnovni nabor operacij za izvajanje lokalnih in oddaljenih ukazov v ukazni lupini (angl. shell), nalaganje in prenašanje datotek ter ostale podporne funkcionalnosti, čakanje uporabnika na vnos, paralelno izvajanje na več vozlišči, upravljanje z napakami (error handling), itd.

Tipičen primer uporabe je priprava Pyhon modula, ki vsebuje eno ali več funkcij in njegovo izvajanje preko *fab* ukaza v ukazni vrstici; izredno prilagodljiva komanda, ki omogoča izvajanje kombinacije zaporedja pripravljenih ukazov na poljubni množici oddaljenih sistemov.

4.2.3 Pamariko

Paramiko [17] je Python izvedba SSHv2 protokola za zagotavljanje varne povezave (enkriptirane in avtenticirane) do oddaljenih sistemov. Zagotavlja funkcionalnost SSH serverja kot klienta. Ime izhaja iz esperanta (umenti pomožni sporazumevalni jezik) in pomeni »paranoičen prijatelj«. Knjižica je v celoti izvedena v Python jeziku (razen funkcionalnosti za kriptografijo, kjer se uporablja C-implementacija; PyCrypto) in izdana pod GNU LGPL odprtokodno licenco.

4.3 Razvoj

ChessCloud storitev je razdeljena na dva neodvisna dela. To sta analitična gruča, namenjena analiziranju šahovskih pozicij, ter spletna aplikacije, ki ponuja analizo iger in igranje z računalniškim strojem. Pri razvoju gruče so uporabljene javanske tehnologije, ki omogočajo zasnovo skalabilnih in visoko zmogljivih oblačnih sistemov, pri spletni aplikaciji pa tehnologije, ki omogočajo razvoj modernega in odzivnega spletnega vmesnika.

4.3.1 PlayFramework

PlayFramework [18] je odprtokodno okolje za izdelavo spletnih aplikacij in omogoča razvoj v Java in Scala programskem jeziku. Arhitekturna zasnova je osnovana na podlagi MVC (angl. model-view-controller) modela in ponuja preprosto okolje brez stanja (angl. stateless), s pomočjo katerega poskuša povečati produktivnost ravzijalcev. Osnovni koncpet okolja se naslanja predvsem na uporabo paradigme »convention over configuration«, ki postavlja preproste in standardizirane pristope pred nastavljivostjo, pri čemer pa ne izgublja na prilagodljivosti.

PlayFramework ponuja predvidljivo in minimalno porabo virov (procesorska moč, spomin, niti) visoko skalabilnih aplikacij. Glavne značilnosti okolja so :

- ***Prijazno do razvijalcev (angl. Developer friendly)***
  - Vroča menjava kode in takojšnje osveževanje (angl. *Hot swap in Hit refresh workflow*).
  - Močna ukazna lupina in orodja.
  - Močno tipiziran jezik (angl. *Strongly typed language*).
- ***Predvidljivo skaliranje (angl. Scale predicatbly)***
  - Spletni nivo brez stanja (*Stateless Web Tier*).
  - Vhod/izhod brez zapore (*Non-blocking I/O*).
  - Asihrono procesiranje z uporabo Akka okolja.
- ***Zanesljivost in hitrost***
  - Prevajanje kode.
  - Izvajanje na javanskem virutalnem stroju (Java Virtual Machine)
  - Netty – ne blokirajoči vhodno/izhodni (NIO) klient-server ogrodje.
- ***Razvoj modernih spletnih in mobilnih aplikacij***
  - RESTful po dizajnu.
  - Podpora JSON formatu.
  - Namenski prevajalniki za *CoffeeScript*, *LESS*, itd.
  - Podpora *Websockets*, *Comet*, *EventSource* protokolom.

- Obsežna podpora podatkovnih baz tipa *NoSQL* in *Big Data.*

- ***Velik in agilen ekosistem***
  - Aktivna skupnost razvijalcev in uporabnikov.
  - Podpora Maven repositorijem.
  - Veliko število prilagojenih vtičnikov (angl. plugins).

4.3.2 Akka

Akka [19] je odprotokodni nabor orodij in izvajalnik kode, namenjen gradnji sočasnih in distribuiranih aplikacij, ki tečejo na javanskih virtualnih storjih (JVM). Akka podpira več programskih modelov za reševenje sočasnosti, vendar poudarja model sočasnosti z uporabo akterjev (angl. actor-based concurrency model), s pomočjo katerih ponuja okolje za gradnjo skalabilnih, odzvinih in robustnih aplikacij. Akterji omogočajo tudi transparentno distribuiranost, kar je osnova za visoko skalabilne sisteme, odporne na morebitne napake. Glavne značilnosti okolja so:

- ***Proprosta uvedba sočasnosti in distribuiranosti***, ki je dosežena z arhitekturo okolja, ki po dizajno omogoča asinhrono in distribuirano komunikacijo.
- ***Visoka zmogljivost*b,** ki je dosežena z minimalnim asihronim dizajnom akterjev.
- ***Odpornost na napake*** katere podpora je vgrajena v sam dizajn okolja in omogoča razvoj sistemov, ki se sami popravljajo (angl. self-healing systems).
- ***Elastičenost in decentraliziranost***, kar omogoča napreden sistem za oddaljen nadzor.
- ***Razširljivost***, ki je dosežena s pomočjo Akka vtičnikov, ki prilagodijo Akka okolje potrebam razvijalcev.

4.3.3 Nginx

Nginx [20] je odprtokodni visoko performančni server in povratni namesntik (angl. reverse proxy) za HTTP, HTTPS, SMTP, POP3 in IMAP protokole. Strežnik je znan po svoji hitrosti, nizki porabi sistemskih sredstev ter zmožnostjo dela z veliko hkratnimi povezavami. Uporablja se ga pa tudi za porazdelitev obremenitev, serviranje statičnih vsebin in kot HTTP predpomnilnik. Za serviranje se uporabjlja asihroni pristop, kar pripomore, da se pod obremenitvijo obnaša veliko bolj predvidljivo kot strežniki, ki uporabljajo nitni ali procesni pristop. Glavne značilnosti so:

- Dodogkovna arhitekutra.
- Glavni in več delovnih procesov (angl. master-slave processing).
- Spreminjanje nastavitev brez prekinitve.

- Povratni namesntnik z možnostjo prepomnenja, izenečevanje obremenitev in redudance.
- SSL in TLS podpora.
- Podopra virtualnim strežnikom.
- Spreminjanje URL naslovov (angl. URL rewrite).
- Vgrajen HTTP gzip modul.

4.3.4 AngularJS

AngularJS [21] je odprtokodno okolje za razvoj dinamičnih spletnih aplikacij, t. i. enostranskih aplikacij (angl. single-page applications). Cilj okolja je poenostaviti razvoj in testiranje s poudarkom na MVC (model-view-controller) arhitekturo in komponentami, ki jih pogosto uporabljajo pri razvoju bogatih internetnih aplikacijah (angl. rich internet applications). S pomočjo vezave podatkov s predstavitvijo (angl. data binding) in injiciranjem odvisnosti (angl. dependency injection) odpravlja potrebo po ponavljajočem pisanju kode (angl. boilerplate code) in tako pospeši razvoj in olajša vzdrževanje. Okolje odpravlja potrebo po pisanju kode za:

- Registracije povratnih klicev (angl. call-back).
- Programsko manipuliranje HTML DOM elementov.
- Prenos in formatiranje podatkov v in iz uporabniškega vmesnika.
- Inicializacijo pogostih funkcionalnosti; na primer že preprosta AJAX funkcionalnost zahteva kar nekaj začetne kode.

Okolje deulje na način, da preišče HTML stran za morebitnimi Angular atributi »ng«, ki jih interpretira kot deklerative aplikaciji, s pomočjo katerih AngularJS omogoča:

- Povezovanje podtakov med modelom in prezentacijo (predstavitvijo).
- Kontrolne strukture za podvajanje, prikazovanje in skrivanje DOM fragmentov.
- Podpora za spletne obrazce in njihovo vrednotenje.
- Dodajanje lastnosti DOM elementom (npr. upravljanje z dogodki).
- Grupiranje HTML fragmentov v neodvisne in ponovno uporabne komponente.

4.3.5 ChessBoardJS in ChessJS

Chessboard.js [22] je odprtokodna šahovska JavaScript knjižica, namenjena izrisu šahovske plošče in interakciji s figurami. Osnovna funkcionalnost je zelo preprosta in ne vsebuje dodatne logike, kot je na primer validacija dovoljenih potez, šahovski stroj, razčlenjevanje PGN zapisa. Kljub temu s pomočjo močnega programskega vmesnika omogoča integracijo z ostalimi knjižicami, ki rešujejo te probleme, kot na primer ChessJS.

ChessJS [23] je odprtokodna šahovska JavaScript knjižica, namenjena generiranju in vrednotenju dovoljenih potez, detekciji šaha (napad na kralja) in detekciji konca igre (šah-mat ali pat). Omogoča pa tudi nadzor poteka igre (trenutna pozicija, zgodovina potez, premiki nazaj in naprej) in razčlenjevanje dveh pogostih šahovkih zapisov; PGN in FEN (glej 2.2.3 in 2.2.4).

4.4 Ostalo

4.4.1 Ganglia

Ganglia [24] je skalabilni distribuirani sistem za spremljanje visoko zmogljivih računalniških sistemov (npr. gruče). Omogoča pregled statistik in zbranih podatkov oddaljeno, preko spletne aplikacije. Rešitev je zanesljiva in robusnta, uporabljena na več tisoč gurčah po svetu in široko podprta na različnih opreacijskih sistemih in arhitekutrah.

Uporabljene podatkovne strukture in algoritmi so razviti na način, da minimizirajo potrebno delo na vozliščih in omogočajo visoko stopnjo sočasnosti in prepustnosti. Za zapis podatkov se uporablja format XML, za njihov učinkovit prenos serializacijski format XDR, za njivo hrambo in vizualizacijo pa RRDtool. Glavne komponente sistema Ganglia:

- **Monitoring proces (gmond)** : namenjen je spremljanju vozlišč. Je večnitni in teče na vsakem vozlišču, ki ga hočemo nadzirati. Glavne funkcionalnosti procesa so:
  - Monitoriranje stanja vozlišča.
  - Objavljanje relevantnih sprememb na vozlišču.
  - Poslušanje ostalih vozlišč s pomočjo unicast ali multicast kanala.
  - Odgovarjanje na poizvedbe o stanju gruče.
- **Meta proces (gmetad)** je zadolžen za periodično zbiranje podatkov, zbranih na vozliščih, za njihovo organizacijo in shranjevanje. Klientom preko TCP vtičnika omogoča poizvedbe in izvoz shranjenih podatkov.
- **Spletna aplikacija:** omogoča pregled nad zbranimi podatki. Omogoča poizvedbe nad zgodovino ali pregled zbranih podatkov v realnem času.

4.4.2 Gattling

Gatling [25] je odprtokodno orodje, ki omogoča obremenitveno testiranje (angl. load testing) ter analizo zmogljivosti spletnih storitev s poudarkom na spletne aplikacije. Orodje podpira

testiranje preko različnih kanalov oziroma protokolov: HTTP, WebSockets, JMS, Server Sent Events.

Orodje je načrtovano v smeri lahke uporabe in vzdrževanja ter visoke zmogljivosti. Arhitektura je asinhrona in sloni na osnovi okolja Akka in Netty, zaradi česar je izvedba virtualnih uporabnikov asinhrona in ne z uporabo namenskih niti, ki zahtevajo veliko več procesorskih in spomniskih virov.

Poglavje 5 Razvoj storitve

To poglavje je namenjeno opisu ChessCloud storitve s stališča uporabnika kot tudi s stališča arhitekturne zasnove in razvoja, s pripadajačimi nefunkcionalnimi zahtevami in rešitvami, ki jih navadno srečamo pri spletnih aplikacijah. V nadaljevanju poglavja pa so opisane glavne komponente sistema, njihova realizacija ter komunikacija med njimi. Glavne komponente sistema so:

- Analitčna gurča: zajema razvoj treh tipov vozlišč (*delivec, proizvajalec, izvajalec)*, ki sestavljajo gručo. Opisana je komunikacija z uporabnikom na strani *proizvajalca,* dodeljevanje in upravljanje na starni *delivca* ter izvrševanje analiz in komunikacija s strojem na strani *izvajalca.*
- Strežniška aplikacija: zajema razvoj strežniškega dela aplikacije z REST vmesnikom, ki omogoča zahteveke za analizo, z uporabo treh tehnologij, in sicer s standardno GET poizvedbo, Comet poizvedbo in Websocket dvosmerno komunikacijo.
- Spletna aplikacija: zajema razvoj enostranske AngularJS aplikacije, kot primer spletnega šahovskega okolja, ki omogoča analizo poljubnih šahovskih pozicij, igranje z računalnikom in pregled šahovskih partij.

To poglavje predstavlja izključno opis in razvoj storitve, medtem ko so postavitev, upravljanje ter obremenitveno testiranje opisane v naslednjem poglavju.

5.1 Opis storitve

Storitev, poimenvana ChessCloud, omogoča pregled in analizo šahovskih partij ter igranje s šahovskim strojem preko spletnega brskalnika. Storitev je sestavljena iz dveh delov, torej iz spletne aplikacije in gruče. Spletna aplikacija ponuja grafično okolje namenjeno uporabnikom, ki poleg analize šahovskih pozicij ponuja tudi dodatne zmožnosti.:

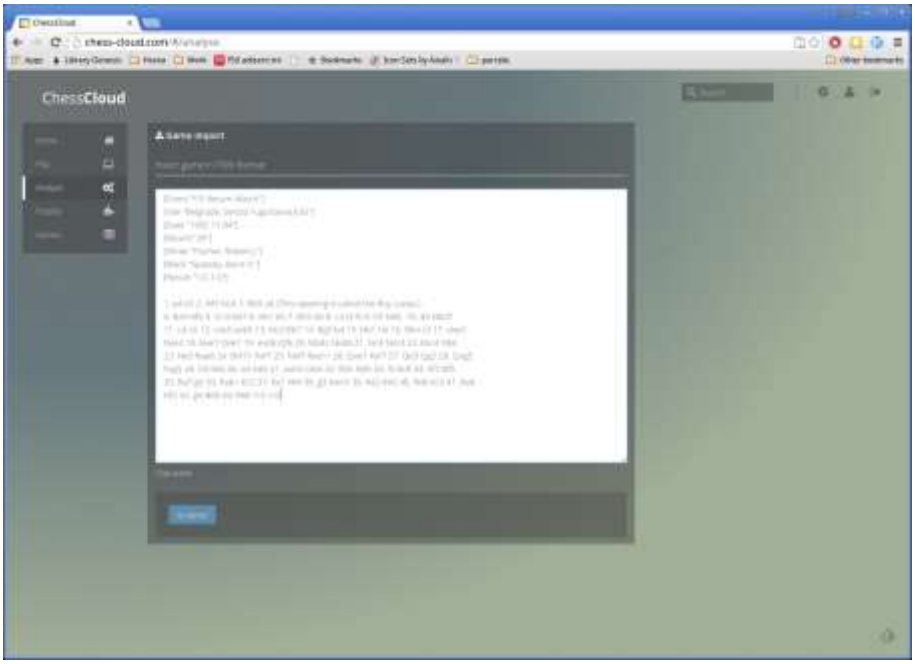
- **Vnos in pregled igre**: storitev omogča vnos igre v PGN zapisu in vnos pozicije v FEN zapisu. Pri vnosu igre lahko uporabnik poljubno navigira med potezami in le-te analizria. Pri vnosu pozicije pa je mogoče vnašati nadaljnje poteze in le-te tudi analizirati.
- **Analiza šahovskih pozicij**: storitev za poljubno pozicijo poda tri najboljše poteze, skupaj z evalvacijo in potjo za naslednjih nekaj potez. Uporabnik ima možnost

pregledati vsako izmed poti na šahovski plošči. Med analizo servis uporabnika sproti obvešča o trenutnem stanju analize, tako da mu ni potrebno čakati na konec, da dobi občutek, kaj se v trenutni poziciji dogaja.

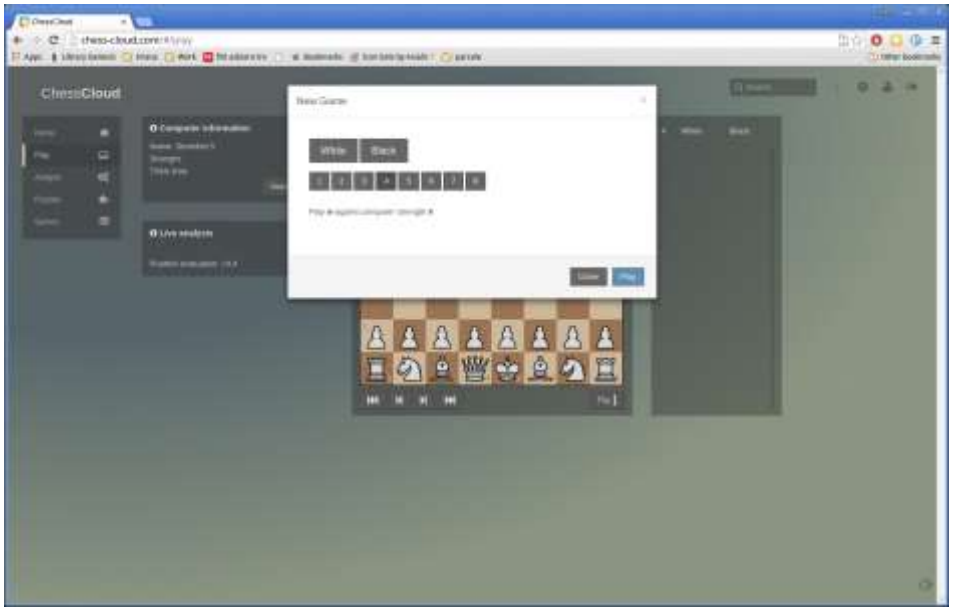
- **Igranje s šahovskim strojem**: storitev omogoča igranje s šahovskim strojem na različnih težavnostnih stopnjah. Uporabnik lahko začne igro iz začetne pozicije s poljubnimi barvami, mogoče pa je igrati tudi iz pogleda analize s trenutno izbrano pozicijo.
- **Pregled shranjenih iger**: trenutna verzija omogoča nalaganje shranjenih iger, ki so namenjene testiranju storitve.
- **REST vmesnik**: storitev ponuja preprost vmesnik, s pomočjo katerega se sistemu pošljejo zahtevki za analize in s tem omogoča uporabo analitične gruče tudi ostalim storitvam (angl. third-party services).

Spodaj so zaslonske maske, ki prikazujejo različne funkcionalnosti storitve. Prva slika prikazuje analizo trenutne pozicije s tremi možnimi nadaljevanji in evalvacijo za vsako izmed njih. Druga slika prikazuje vnos partije v PGN zapisu. V primeru neveljavnega zapisa je uporabnik o tem obveščen. Tretja slika prikazuje začetek igre s šahovskim strojem, kjer je mogoče nastavljati barvo figur in težavnostno stopnjo.

Slika 9: Analiza šahovske igre



Slika 10: Uporabniški vmesnik za vnos partije v PGN formatu



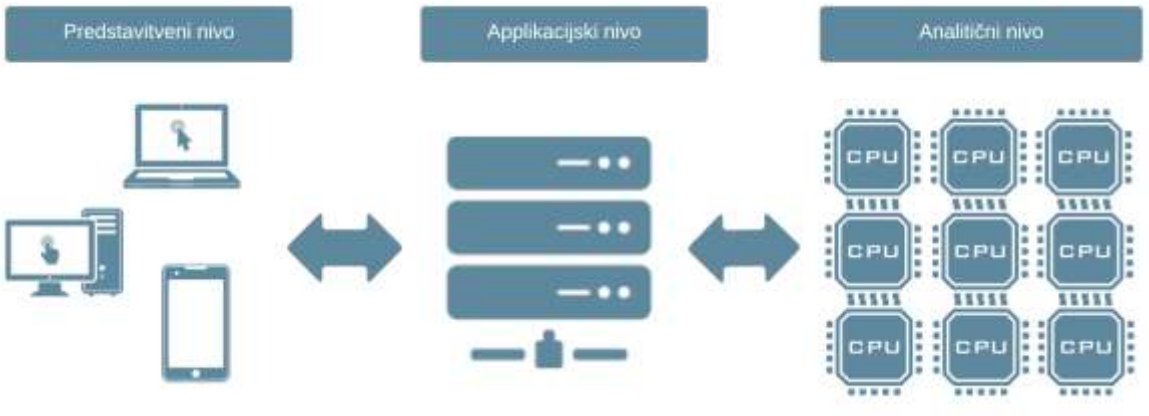
Slika 11: Dialog za pričetek igre z računalniškim strojem

ChessCloud storitev je nameščena v privatnem oblaku (uporabljen je OpenStack) in je dostopna na naslovu http://www.chess-cloud.com. Storitev je javno na voljo in ne zahteva avtentikacije. Podrobnosti namestitve so opisane v poglavju 6. Postavitev in upravljanje.

5.2 Arhitekura

Arhitektura storitve ChessCloud upošteva vzorce in dobre prakse porazdeljenega računalništva. Sorodne funkcionalnosti so združene v komponente, ki so med seboj šibko sklopljene (angl. loosely coupled) in brez stanja (angl. stateless), kar omogoča, da sistem vsako komponento neodvisno podvaja (angl. scale-out) in s tem zagotavlja boljšo odzivnost ter razpoložljivost. Posamezne komponente ali sestavine predstavljajo nivoje, zato lahko govorimo o več nivojski arhitekturi, ki sicer ni tipična »n-tier« arhitektura, saj podatkovni nivo nadomešča analitični nivo, gre pa v principu za enak vzorec. Storitev se tako deli na:

- **Predstavitveni nivo** (angl. Presenation tier) predstavlja aplikacija, ki teče v spletnem brskalniku.
- **Aplikacijski nivo** (angl. Application tier) predstavlja strežniška aplikacija, ki povezuje predstavitveni nivo z gručo.
- **Analitični nivo** (angl. Analysis tier) predstavlja gruča, namenjena analizi šahovskih pozicij.



Slika 12: Večnivojska arhitektura v storitvi ChessCloud

Slika predstvalja posamezne nivoje storitve ChesscCloud. Pomembno pri takšni arthitekturi je, da komunikacijo med nivoji natančno določimo, saj le tako dosežemo popolno delitev med nivoji in s tem neodvisnost med komponentami. Protokol, ki opisuje potek komunikcije na nivoju gruče z aplikacijskim nivojem, je opisan v podpoglavju 5.3.4. Protokol med predstavitvenim nivojem in aplikacijskim nivojem je opisan v poglavju 5.4.1.

Večnivojska arhitekturna zasnova, v primerjavi z eno- in dvonivojsko arhitekutro, prinaša veliko prednosti. Storitev je skalabilna in omogoča serviranje večjega števila uporabnikov. Komponente ali sestavine so med seboj neodvisne, zato jih je mogoče ponovno uporabiti. Izboljšana je zanesljivost storitve, pa tudi vzdrževanje, nadgradnje, razširitve so lažje izvedljive. Kjlub vsem prednostim pa se poveča varnostno tveganje, saj je lahko ranljiva vsaka komponenta posebej kot tudi kompleksnost vzrdrževanja, postavljanja ter nadzorovanja sistema, saj teče na več različnih računalniških virih in omrežjih.

5.2.1 *Nefunkcionalne zahteve*

Poleg poslovnih funkcionalnosti (angl. business requirements) so pri načrtovanju arhitekture pomembne tudi nefunkcionalne zahteve (angl. non-functional requirements), ki se nanašajo na obnašanje storitve v oblaku. Zagotavljanje le-teh je izrednega pomena pri končnih uporabnikih, saj naredi uporabniško izkušnjo kakovostno in konsistento; ne glede na čas in število sočasnih uporabnikov. Spodaj so navedene zahteve, ki jih storitev ChessCloud zagotavlja in se delno nanašajo tudi na lastnosti računalništva v oblaku, opisane v poglavju 3.

- **Skalabilnost:** arhitektura sistema z uporabo neodvisnih komponent brez stanja zagotavlja zmožnost podvajanja in s tem tudi dodajanje novih računalniških virov, kar omogočajo elastične infrastrukture. Vsaka komponenta v poljubnem nivoju ima zmožnost komuniciranja s poljubno komponento naslednjega nivoja, kar je doseženo z uporabo programskih vzorcev, tipičnih za oblačne in distribuirane sisteme; porazdelitev obremenitve (angl. load-ballancing pattern), vzorec objave in naročanja (angl. publish-subscribe pattern), vzorec distribuiranih izvajalcev (angl. distributed worker pattern) itd.
- **Dostopnost:** Storitev je dostopna preko internetnega omrežja na domeni chess-cloud.com.
- **Zanesljivost:** Integriteta podatkov in konsistenca sistema ostajata nespremenjena celo v primeru povečane obremenitve ali izpada določene komponente.
- **Razporožljivost**: zaradi podvajanja virov na vsakem nivoju sistem ostaja delujoč tudi v primeru, da pri eni izmed komponent pride do napake oziroma postane neodzivna.
- **Razširljivost**: Komponente so nedovisne in izolirane. Če se protokol ne spremeni, se lahko funkcionalnosti dodajajo brez težav in vpliva na ostale nivoje. Takšna arhitektura omogoča tudi lažje vzdrževanje in nadgrajevanje.
- **Odzivnost**: Sistem ostaja odziven tudi v primeru, da v trenutku ni na voljo prostih izvajalskih zmožnosti. V tem primeru gre zahteva v čakalno vrsto, o čemer je uporabnik obveščen.
- **Varnost**: Edina vstopna točka v sistem je REST vmesnik, kjer se vsi možni vnosi preverijo. Ker je sistem postavljen v privatnem oblaku, je pred vstopom v sistem

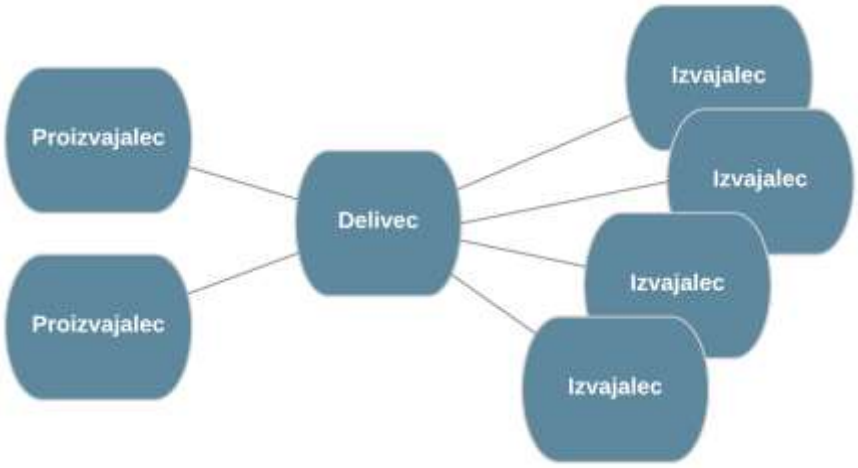
varnost zagotovljena na več točkah v privatni infrastrukturi. Dostop je mogoč samo z uporabo domene preko domenskega strežnika, ki zunanji IP in domeno preslika v notranjega, kar z drugimi besedami pomeni, da so virtualizirani računalniški viri direktno dostopni samo znotraj internega omrežja.

5.3 Analitična gruča

Za razvoj analitične gruče je uporabljena platforma »Akka« in razširitev »Akka Cluster«, ki omogoča razvoj decentralizirane gruče, odporne na napake (angl. fault-tolerant) in enotokčovna ozka grla (angl. single point of bottleneck). Članstvo v gurči in komunikacija potekata po principu vsak z vsakim (angl. peer to peer) z uporabo Gossip protokola[3], pri uporabi katerega vozlišča širijo informacije ali obvestila s poljubnimi sosednjimi vozlišči in nekako posnemajo širjenje govoric v družbi.

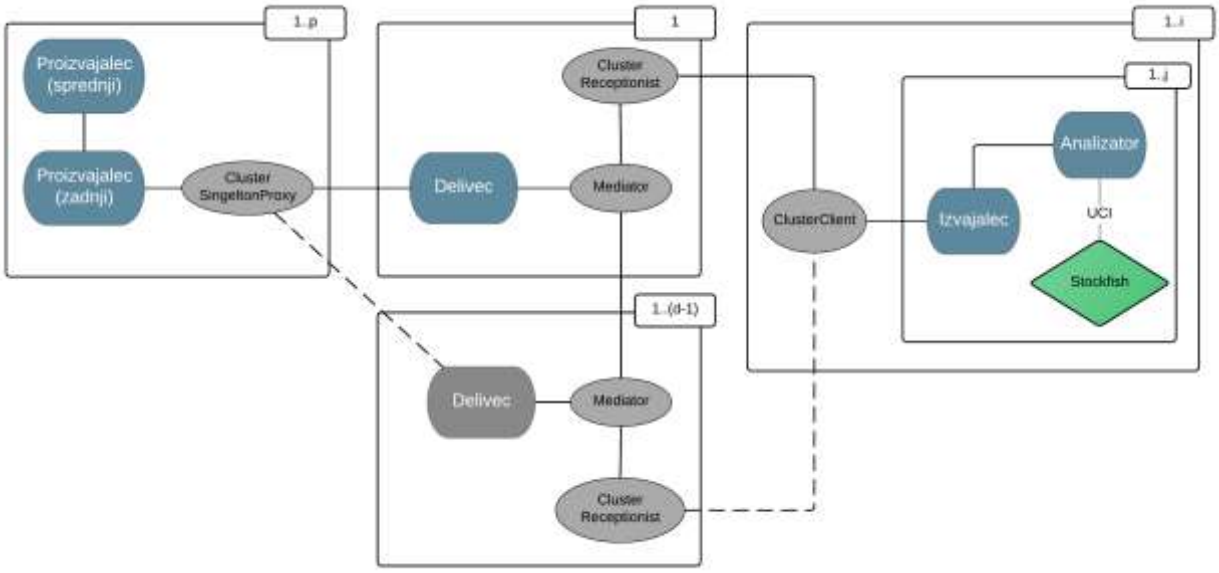
Gruča sestoji iz treh tipov vozlišč. Sestavljajo jo *proizvajalec*, *delivec* in *izvajalec*. Kot nakazujejo imena, je *proizvajalec* zadolžen za proizvodnjo novih zahtev (spodbujen s strani uporabnika oziroma predstavitvenega nivoja), *delivec*, ki upravlja s čakalno vrsto in deli dela *izvajalcem*, ki so zadolženi, da ga opravijo in o tem obvestijo ostala vozlišča. Pri gruči se uporablja programski vzorec »*distribuirani izvajalci*« (angl. distributed worker pattern), ki je izvedenka vzorca »*delivec-izvajalec*« (angl. master-worker pattern), kjer so izvajalci porazdeljeni v omrežju in so, s strani *delivca*, obveščeni o novem delu, ki je na voljo. Po obvestilu mora za delo *izvajalec* zaprositi sam. S tem je rešen problem preopremenjenih izvajalcev in je nadzor nad zahtevami centraliziran ali osrediščen na strani *delivca*. Slednje se lahko združi s sistemom za samodejnim prilagajanje *izvajalcev* (angl. scale-out), ki *delivcem* omogoča dinamično dodajanje in odstranjevanje *izvajalcev*.

[3] https://en.wikipedia.org/wiki/Gossip_protocol



Slika 13: Primer analitične gurče

Slika prikzuje primer ChessCloud gruče z dvema *proizvajalcema*, ki sta povezana z *delivcem*, in množico *izvajalcev*, ki so zadolženi za analizo šahovskih pozicij. Slednji so zadolženi za analizo šahovskih pozicij, za kar uporabljajo Stockfish šahovski stroj.



Slika 14: Sestava analitične gruče in potek komunikacije

Slika podrobneje prikazjuje sestavo analitične gurče in komunikcijo med komponentami ali sestavinami, ki so podrobnje opisane v nadaljevanju. V gruči je vedno glavno samo eno vozlišče tipa *delivec*, medtem ko so ostala v pripravljenosti, da prezvamejo vodstvo, v kolikor

to postane neodzivno. To omogočata Akka komponenti *ClsuterSingeltoManager,* ki spremlja stanja, in *ClusterSingletonProxy,* preko katere je proizvajalcu omogočeno, da vedno komunicira z glavnim delivcem. Podrobnosti *delivca* so opisane v 5.3.1. Izvajalska vozlišča niso del Akka gruče, vendar z njo komunicirajo s pomočjo komponent *ClusterClient* in *ClusterReceptionist.* Slednji v sodelovanju s komponento *DistributedPubSubMediator* realizira porazdeljen sistem objavaljanja in naročanja (angl. distributed publish-subscribe system). *Izvajalsko* vozlišče lahko vzpostavi več *izvajalcev*, kjer vsak neodvisno opravlja delo analiziranja šahovskih pozicij s pomočjo šahovskega stroja Stockfish preko UCI vmesnika. Akter, ki omogoči izvrševanje nalog, se imenuje *Analizator,* neposredno pa ga upravlja *delivec.* Podrobnosti *izvajalca* so opisane v 5.3.3. Kot kaže slika, je *proizvajalec* razdeljen na sprednji in zadnji del, kjer je prvi namenjen komunikaciji z zunanjim svetom (uporabniki), medtem ko zadnji del skrbi za komunikcijo z gručo oziroma z *delivcem*. Podrobnosti *proizvajalca* so opisane v 5.3.3.

5.3.1 Delivec

Glavni del analitične gruče, ki nadzoruje celotni potek, je *delivec*, ki delo pridobiva od *proizvajalcev* in ga po določenih pravilih dodeljuje registriranim *izvajalcem*. *Delivec* skrbi za to, da se delo opravi na dodeljenem *izvajalcu* oziroma v primeru napake dodeli delo novemu *izvajalcu.*

Izvedena rešitev omogoča, da hkrati teče več *delivcev*, vendar je vedno aktiven (angl. master) samo eden, kar pomeni, da sam nadzira vso komunikacijo s *proizvajalci* in *izvajalci*. Ostali *delivci* v sistemu so v tem času neaktivni, a v pripravljenosti, da eden izmed njih prevzame vodstvo v primeru, da trenutno aktivni *delivec* postane neodziven. Delo, ki se trenutno opravlja, se nadaljuje, rezultat tega pa se sporoči novemu glavnemu *delivcu*. Vzorec edinca (angl. singleton pattern) omogoča *ClusterSingletonManager* komponenta, na voljo v Akka okolju, ki dovoljuje, da je v gruči aktiven natanko en akter določenega tipa. Izsek kode, odgovorne za uvodno nastavitev gruče prikazuje zagon *delivca* z uporabo *ClusterSingletonManager* komponente.

```
// Master initialization
def startMaster(conf : Config): Unit = {
  val system = ActorSystem("ClusterSystem", conf)
  system.actorOf(ClusterSingletonManager.
          props(Master.props(workTimeout),
          "active",PoisonPill, Some("backend")), "master")
}
```

Izsek 8: Uvodna nastavitev *delivca*

Delivec je izveden kot Akka akter, ki deluje kot končni avtomat z enim stanjem; *receive.* V tem stanju sprejema zahteve *proizvajalcev* in jih dodeljuje *izvajalcem* po sistemu »prvi pride, prvi postrežen« (angl. FCFS - first-come, first-served). Opravljeno delo tudi spremlja in v primeru napake oziroma neodzivnosti *izvajalca,* tega izloči in ponovno dodeli delo prostemu *izvajalcu*. Spodaj je izsek kode *delivca*, ki prikazuje glavne dele akterja.

```
object Master {

  // Master initialization properties
  def props(workTimeout: FiniteDuration): Props =
          Props(classOf[Master], workTimeout)

  // Internal data objects
  private sealed trait WorkerStatus
  private case object Idle extends WorkerStatus
  private case class Busy(id: String, d: Deadline) extends WorkerStatus
  private case class WorkerState(ref: ActorRef, status: WorkerStatus)
  private case object CleanupTick
}

class Master(workTimeout: FiniteDuration) extends Actor with ActorLogging {

  // ... Master initialization ...

  // Initialize mediator, to listen to Workers
  val mediator = DistributedPubSubExtension(context.system).mediator
  ClusterReceptionistExtension(context.system).registerService(self)


  // Main state
  override def receive: Receive = {
    case MasterWorkerProtocol.RegisterWorker(workerId) => ...
    case MasterWorkerProtocol.WorkerRequestsWork(workerId) => ...
    case MasterWorkerProtocol.WorkIsDone(workerId, workId, result) => ...
    case MasterWorkerProtocol.WorkFailed(workerId, workId) => ...
    case work: MasterFrontendProtocol.Work => ...
    case CleanupTick => ...
  }

  // ... support methods ...
  // ... cleanup timer ...
  // ... exception handling ...
}
```

Izsek 9: Izvedba akterja *delivec*

5.3.2 Proizvajalec

Proizvajalec v sistemu ChessCloud teče na aplikacijskem nivoju na strežniškem delu spletne aplikacije, ki teče v okolju Play 2.0. To je doseženo tako, da se ob zagonu aplikacije uvodno

nastavi nov sistem akterjev, ki se je sposoben povezati na analitično gručo. Komunikacija z gručo poteka s pomočjo *ClusterSingletonProxy* komponente, ki omogoča, da proizvajalec vedno komunicira z glavnim *delivcem*.

Proizvajalec je razdeljen na dva dela:

- Sprednji del je zadožen za komunikacijo z uporabnikom.
- Zadnji del je zadolžen za komunikacijo z gručo.

Zadnji del je izveden kot akter, ki je zadolžen za posredovanje nalog glavnemu *delivcu.* To doseže s pomočjo komponente *ClusterSingletonProxy,* ki omogoča, da komunikacija vedno poteka z *glavnim delivcem.* V primeru napake ali neodzivnosti *delivca* o tem obvesti pošiljatelja dela. Spodnji izsek prikazuje izvedbo akterja.

```
class Frontend extends Actor {

  // Initialize master proxy
  val masterProxy = context.actorOf(ClusterSingletonProxy.props(
    singletonPath = "/user/master/active",
    role = Some("backend")),
    name = "masterProxy")

  // Forward work to master
  def receive = {
    case work =>
      implicit val timeout = Timeout(5.seconds)
      (masterProxy ? work) map {
        case MasterProtocol.Ack(_) => Ok
      } recover { case _ => NotOk } pipeTo sender()
  }
}
```

Izsek 10: Izvedba akterja *proizvajalca (sprednji del)*

Sprednji del, zadolžen za komunikacijo z uporabnikom, je akter, ki je ustvarjen pri vsaki novi povezavi z uporabnikom in komunicira z akterji, ki so neposredno povezani z uporabniki; *AnalysisApiActors* (glej 5.4.1). Njegova naloga je zahtevati delo od zadnjega dela *proizvajalca* in komuniciranje z *izvajalcem,* kateremu je delo dodeljeno. O rezultatih mora obveščati uporabnikovega akterja. Prikazan je izsek izvedbe prednjega dela *proizvajalca*, ki vsebuje tri stanja:

- Stanje čakanja, kjer akter čaka na zahtevo uporabnika. Ko pride zahteva, jo pošlje naprej in preide v naslednje stanje.

- Stanje čakanja na *izvajalca*, kjer akter čaka na prostega izvajalca. Ko se mu ta oglasi, mu pošlje zahtevo za analizo in preide v naslednje stanje. V primeru napake o tem obvesti uporabnika in preide nazaj v začetno stanje.
- Stanje analize, kjer akter čaka na rezulate in jih pošilja naprej. Omogočena pa je tudi povratna komunikacija, kjer lahko uporabnik pošilja komande *izvajalcu* (v primeru WebSocket-a)

```
class ProducerActor (out:ActorRef) extends Actor with ActorLogging{

  // Idle state - wait for new request
  override def receive = {
    case af : AnalyseFen => ...
  }

  // Waiting state - wait for worker, start analysis
  def waitForWorker (analyseFen: AnalyseFen): Actor.Receive = {
    case Frontend.Ok => ...
    case Frontend.NotOk => ...
    case WorkerAssigned (worker : ActorRef) => ...
    case ReceiveTimeout => ...
  }

  // Analysis state - wait for results, send API messages (i.e. stop)
  def analysisStarted (worker: ActorRef) : Receive = {
    case r: AnalyserProtocol.ResultUci => ...
    case bm: AnalyserProtocol.BestMove => ...
    case f: AnalyserProtocol.Finished => ...
    case m: Message => ...
  }
}
```

Izsek 11: Izvedba akterja *proizvajalca (zadnji del)*

5.3.3 Izvajalec

Izvajalci so zadolženi za izvajanje naročenega dela, tj. evalvacija šahovske pozicije. Za razliko od *proizvajalcev* in *delivcev izvajalci* strogo gledano niso del gruče, vendar z njo komunicirajo. Razlog je v tem, da sistem dopušča, da *izvajalci* niso stabilni oziroma zanesljivi; »pusti, da pade« princip (angl. let-it-fail). Ker niso del gruče, ne morejo, ko pride do napake, vplivati nanjo (npr. deljenje gruče ali nepotrebno preverjanje prisotnosti). V primeru napake *izvajalec* ne dobi nadaljnjega dela, ker za njega ne zaprosi, nedokončano delo pa *delivec* dodeli novemu *izvajalcu*.

Komunikacija z gručo se vzpostavi s komponento *ClusterClient* na strani izvajalca in *ClusterReceptionist* na strani delivca. Slednji v gruči vzpostavi porazdeljen sistem objavaljanja in naročanja (angl. distributed publish-subscribe system) in tako omogoči, da se

izvajalci registrirajo pri delivcu in zaprosijo za delo. Dogodki se med komponentama pošiljajo in prejemajo asinhrono, na enak način, kot je to značilno v Akka okolju. Izsek prikazuje uvodno nastavitev aktor sistema *izvajalcev*, vzpostavitev *ClusterClient* komponente in nastavitev željenega števila *izvajalcev.*

```
// Workers initialization
def startWorker(conf :Config): Unit = {

  // Initialize Actor system
  val system = ActorSystem("WorkerSystem", conf)

  // Initialize ClusterClient with initial contact points
  val contactPoints = immutableSeq(conf.getStringList("contact-points"))
  val initialContacts = contactPoints.map {
    case AddressFromURIString(addr) ⇒
      system.actorSelection(
            RootActorPath(addr) / "user" / "receptionist")
  }.toSet

  val clusterClient = system.actorOf(
          ClusterClient.props(initialContacts), "clusterClient")

  // Create workers (based on configuration)
  val size = conf.getInt("worker.size")
  for (num <- 0 until size)
  {
    system.actorOf(Worker.props(
          clusterClient, Props[WorkExecutor]), "worker-"+(num+1))
  }
}
```

Izsek 12: Uvodna nastavitev *izvajalcev* z uporabo *ClusterClient* komponente

Izvajalec je akter, ki ima tri stanja:

- Stanje pripravljenosti, kjer *izvajalec* čaka na obvestilo *delivca,* da je novo delo pripravljeno, na katerega se lahko se odzove, da je razpoložjiv. Če ga *delivec* izbere, mu je dodeljeno delo, ki ga posreduje *analizatorju*.
- Stanje čakanja na rezultat, kjer *izvajalec* čaka na *analizatorja,* da zaključi. V tem času se ne odziva na obvestila *delivca.*
- Stanje pošiljanja rezultata, kjer *izvajalec* obvesti *delivca* o končanem delu in preide v stanje pripravljenosti. Hkrati še sporoči *delivcu,* da je pripravljen prevzeti novo delo, če je to na voljo.

```
object Worker {

  // Worker initialization properties
  def props(clusterClient: ActorRef, workExecutorProps: Props,
            registerInterval: FiniteDuration = 10.seconds): Props =
             Props(classOf[Worker], clusterClient, workExecutorProps,       registerInterval)

  // Internal data objects
  case class WorkComplete(result: Any)
}

class Worker(clusterClient: ActorRef,
             workExecutorProps: Props, registerInterval: FiniteDuration)
  extends Actor with ActorLogging {

  // ... Worker and WorkExecutor initialization ...

  // ... Register task – register worker in cluster ...

  // ... Supervisor strategy – Executor exceptions recovery ...

  // Idle state - try to get some work
  def idle: Receive = {
    case WorkIsReady => ...
    case Work(workId, job) =>  ...
  }

  // Working state - wait for complete
  def working: Receive = {
    case WorkComplete(result) => ...
  }

  // Completed state – sync with Master (repeat if neccessery)
  def completed (result: Any): Receive = {
    case Ack(id) if id == workId => ...
    case ReceiveTimeout => ...
  }

  // ... support methods ...

  // ... exception handling ...
}
```

Izsek 13: Izvedba akterja *izvajalec*

Izvajalec sam ne opravlja analize, ampak to delo posreduje svojemu *analizatorju AnalyseActor*, ki je zadolžen za vzpostavitev povezave s šahovskim strojem in komuniciranje z njim preko UCI vmesnika. S to delitvijo je omogočeno, da *izvajalec* nadzoruje izvrševalskega akterja in v primeru nepredvidene napake, le-tega ponovno vzpostavi. Izsek prikazuje izvedbo *analizatorja*, ki ima dve stanji:

- Stanje pripravljenosti, kjer čaka na zahtevek za analizo. Ko ga prejme, se poveže s *proizvajalcem,* ga obvesi, da je na voljo in preide v naslednje stanje.
- Stanje dela je stanje, kjer dvosmerno komunicira s *proizvajalcem*. Od njega prejema zahtevke, pošilja pa vmesne in končne rezultate analize.

```
class AnalyseActor extends Actor with ActorLogging{

  // ... Initialization ...

  // Initialize communication with Stockfish
  val exe = new CommandExecutor(stockfishPath,
    out => ... // ... Standard output stream handling (parse)
    err => ... // ... Error stream handling
    code=> ... //... Program finish code
  )

  // Idle state - wait for work
  def receive = {
          case AnalyseJob (producer: ActorRef) => ...
  }

  // Working state - 2-way communication with producer
  def working (worker: ActorRef, producer: ActorRef) : Receive = {

    case Start (fen: String, skill: String, time: Duration) => ...
    case Stop => ...
    case SendInfo => ...
    case result : Result => ...
    case finished : Finished => ...
    case ReceiveTimeout => ...
  }

  // ... support methods ...

}
```

Izsek 14: Izvedba akterja *analizator*

Vsak akter tiap *analizator* je zadolžen za svoj primerek šahovskega stroja, ki teče v ločenem procesu. Akter se sporazumeva s procesom asinhrono (s pomočjo dogodkov, ki se vnašajo v vrsto), preko standardnega vhoda/izhoda s pomočjo UCI vmesnika. Spodaj je izsek izvedbe komponente *CommandExecutor*, ki vzpostavi proces v okolju Scala, z uporabo »scala.sys.process« knjižice in preusmeri podatkovne tokove v posredovane anonimne funkcije, ki so izvedene v *analizatorju* in so namenjene razčlenjevanju izhodnega UCI vmesnika.

```
class CommandExecutor (command:String, out:String=>Unit, err:String=>Unit, onExit:Int=>Unit) {

  val inputStream = new SyncVar[OutputStream];

  // Initialize OS process and redirect outputs
  val process = Process(command).run(
    new ProcessIO(
      stdin => inputStream.put(stdin),
      stdout => Source.fromInputStream(stdout).getLines.foreach(out),
      stderr => Source.fromInputStream(stderr).getLines.foreach(err)));

  def write(s:String):Unit = synchronized {
    inputStream.get.write((s + "\n").getBytes)
    inputStream.get.flush
  }

  def close():Unit = {
    inputStream.get.close
  }
}
```

Izsek 15: Izvedba komponente *CommandExectuor*

Analizator podpira večji del UCI protokola:

- »uci«, »ucinewgame«: Vzpostavitev šahovskega stroja in nove igre.
- »setoption« : Nastavljanje težavnostne stopnje »Skill Level« in konfiguracija za hkratno preiskovanje več najboljših potez »MultiPV«.
- »position« : Nastavljanje pozicije v FEN zapisu.
- »go« : Zagon iskanja; omejeno časovno in glede na globino.
- »stop« : Ustavitev iskanja; v primeru, da uporabnik prekliče analizo.
- Razčlenjevanje izhoda v času analize : »info« in »bestmove«.

Sodaj je prikazan izsek izvedbe UCI protokola, ki omogoča razčlenjevanje UCI vhodov (angl. parsing) in oblikovanje UCI vhodov (angl. formating).

```
// UCI Commands
trait UCICommand { def command: String }
case class Uci () extends UCICommand {...}
case class UciNewGame () extends UCICommand {...}
case class Position (type: String, pos: String) extends UCICommand {...}
case class SetOption (name: String, v: String) extends UCICommand {...}
case class Go (depth: Int, time: Int) extends UCICommand {...}
case class Stop extends UCICommand {...}

// UCI Responses
trait UCIResponse
case class MultiInfo (depth: Int, time: Int, nodes: Int, moveEvaluations: List[MoveEvaluation])
case class BestMove (move: String, ponder: String) extends UCIResponse
case class Info (depth: Int, time: Int, nodes: Int, multipv: Int, moveEval: MoveEvaluation) extends
UCIResponse

// Dataobjects
case class Score (scoreType: ScoreType.ScoreType, var value: Int)
case class MoveEvaluation (score: Score, moves: String)
object Score {
  val ScoreRegex = """(\w+) (-?\d+)""".r
  def apply (s: String) : Score = /* use regex to parse */
}
object Info {
  val InfoRegex = """info depth (\d+) seldepth \d+ score (\w+ -?\d+).* nodes (\d+) nps \d+ time
(\d+) multipv (\d+) pv (.*)""".r
  def apply (s: String) : Option[Info] = /* use regex to parse */
}
object BestMove {
  val BestMoveRegex = """bestmove (.*) ponder (.*)""".r
  def apply (s: String) : BestMove = /* use regex to parse */
}
```

Izsek 16: Izvedba UCI protokola

5.3.4 Protokol

Sporočanje v gruči poteka od *proizvajalca* preko *delivca* do *izvajalca* in nazaj. Ko uporabnik poda zahtevo po analizi šahovske pozicije, le-to dobi *proizvajalec*, ki je vstopna točka v analitično gručo. Nato komunikacija poteka v naslednjih korakih:

1. *Proizvajalec* pridobi zahtevo, pripravi novo delo in ga pošlje glavnemu *delivcu*. Zahteva se prepošlje od sprednjega dela *proizvajalca*, ki komunicira z uporabnikom, do zadnjega dela *proizvajalca*, ki posreduje zahtevo v gručo glavnemu *delivcu*.
    a. *Delivec* odgovori z ACK (delo je sprejeto) ali ERR (napaka).
    b. *Izvajalec* o statusu zahteve obvesti uporabnika.
2. *Delivec* obvesti *izvajalce*, da je novo delo na voljo; *WorkIsReady*.

   a. *Delivci* zaprosijo za delo (*RequestWork*).
   b. Delo je dodeljeno *delivcu:* prvemu delivcu po principu FCFS (first-come, first-served).

3. *Izvajalec* pošlje zahtevo *izvrševalcu* (akter, ki je del *izvajalca* in zadolžen za opravljenje dela), ki obvesti *proizvajalca*, da je bil dodeljen za opravljanje zahtevanega dela (*WorkerAsigned*). Tako se vzpostavi dvosmerna komunkacija med *izvajalcem* in *proizvajalcem*.
   a. *Proizvajalec* obvesti *izvrševalca*, da naj prične z analizo (*Start*). V primeru, da ni poziva za izvajanje analize, *izvajalec* zaključi z delom.
   b. *Analizator* nastavi šahovski stroj, posreduje pozicijo in sproži iskanje.
   c. *Analizator* sproti sporoča trenutno stanje analize; *ResultUCI*.
   d. Ob koncu analize *analizator* o tem obvesti *proizvajalca* in *izvajalca; WorkCompleted*.

4. Ko je delo končano, *izvajalec* o tem obvesti *delivca*; *WorkCompleted*. Ob prejetju spročila *delivec*, posodobi interno stanje *izvajalcev*, ki ga uporabljajo za obveščanje o novem delu. Poleg tega izvajalec zaprosi za novo delo, če le-to obstaja.

Izsek prikazuje protokol za sporočanje v analitični gruči, kjer so navedeni vsi tipi obvestil, ki se uporabljajo za komunikacijo med vozlišči skupaj s podatkovnimi strukturami, ki se pri tem izmenjujejo.

```
object MasterFrontendProtocol {
  case class Work(workId: String, job: Any)
  case class WorkResult(workId: String, result: Any)
  case class Ack(workId: String)
}

object MasterWorkerProtocol {
  // Master -> Worker
  case object WorkIsReady
  case class Ack(id: String)

  // Worker -> Master
  case class RegisterWorker(workerId: String)
  case class WorkerRequestsWork(workerId: String)
  case class WorkIsDone(workerId: String, workId: String, result: Any)
  case class WorkFailed(workerId: String, workId: String)

}

object AnalyserProtocol {
  // Analysis job (send producer ref)
  case class AnalyseJob extends Job

  // Producer -> WorkExecutor (commands)
  case class WorkerAssigned (worker : ActorRef)
  case class Start (fen: String, skill: String, duration: Duration)
  case class Stop()
  case class Finished()
  case class Ack()

  // WorkerExecutor -> Producer (results)
  case class Result (evaluation: Float, move: String, line: String)
  case class ResultUci (info: MultiInfo)
  case class BestMove (move: String)
}
```

Izsek 17: Protokol za komunikacijo v analitčni gurči

5.4 Spletna aplikacija

Del storitve, ki je namenjen interakciji z uporabnikom, je spletna aplikacija. Spletna aplikacija je zadolžena, da na eni strani komunicira z analitično gručo, na drugi strani pa z uporabniki preko uporabniškega vmesnika. Aplikacija je sestavljena iz strežniškega dela, razvitega v Play 2.0 okolju in AngularJS enostranske spletne aplikacije (angl. single page application). V nadaljevanju je opsian vsak del posebej.

5.4.1 Strežnik

Strežniški del aplikacije je, poleg serviranja statičnih materialov (JavaScript datoteke in knjižice, CSS stili, HTML zaslonske maske, slike itd.), zadolžen za zagotavljanje vmesnika,

preko katerega je moč zahtevati analizo poljubne šahovske pozicije. Omogočeni so trije načini:

- **GET zahteva**: omogoča standardno zahtevanje operacije (analize) s strani strežnika. Zahteva se zaključi, koje analiza končana katere rezultat je v odgovoru – vmesnih rezultati analize se sproti ne pošiljajo. Pozicija in težavnostna stopnja se sporočita s pomočjo poizvedbenih paramterov. Url vstopne točke : /analyse?fen=<fen>&skill=<skill>.
- **Comet**: omogoča sprotno obveščanje o stanju analize, in sicer na način, da uporabniška aplikacija odpre kanal, preko katerega strežnik sporoča stanje. Pozicija in težavnostna stopnja se sporočita s pomočjo poizvedbenih paramterov. Naslov vstopne točke : /comet?fen=<fen>&skill=<skill>.
- **WebSocket**: omogoča vzpostavitev dvosmerne komunikacije med strežnikom in spletno aplikacijo, preko katere aplikacija podaja nove zahteve za analizo, strežnik pa sporoča rezultate in vmesna stanja. Podatki in ukazi se pošiljajo v JSON zapisu. Naslov vstopne točke : /ws.

Spodnji izsek prikazuje izvedbo krmilnika v Play okolju, ki omogoča opisane načine komunikacije.

- *WebSocket.acceptWithActor* omogoča vzpostavitev WebSocket kanala, z uporabo akterja.
- *Ok.chunked (enumerator &> Commet(callback))* omogoča razčlenjen odgovor s pomočjo enumeratorja, ki se ga pošlje akterju. JSON vnosi v enumerator se s pomočjo Comet funkcionalnosti zavijejo v JavaScript klic, ki se kliče na uporabniški strani.
- *Action.async* omogoča asihrono obdelavo poizvedbe. Ko je rezultat poizvedbe na voljo, se le-ta pošlje uporabniku, medtem pa uporabnikova poizvedba čaka.

```
object AnalysisApiController extends Controller {

  // ... Initialization  ...
  // ... Support methods  ...

  // Websocket endpoint (/ws)
  // Open channel using WebSocket actor for 2-way communication
  def ws = WebSocket.acceptWithActor[String, String] {
    request => out => ApiActor.propsWebSocket (out)
  }

  // Comet endpoint (/comet)
  // Create chunked response using Comet functionality
  // (call parent.update JS function)
  def comet () = Action { implicit request =>
    // ... prepare ...
    val analyseFen:AnalyseJob = ... // fen, skill
    val enumerator = Concurrent.unicast[String]
      (onStart = channel => {
        val actor = Akka.system.actorOf
          (ClusterActor.propsChuncked(channel))
        actor ! analyseFen
    })
    Ok.chunked(enumerator &> Comet(callback="parent.update"))
  }

  // GET endpoint (/analysis)
  // Open channel using WebSocket actor for 2-way communication
  def analyse(fen:String) =  Action.async {
      // ... prepare ...
      val analyseFen:AnalyseJob = ... // fen, skill
      val asyncActor = Akka.system.actorOf(Props[AsyncActor])
      (asyncActor ? fenFixed).map(s => Ok(s.toString))
  }
}
```

Izsek 18: Izvedba krmilnika za streženje zahtev za analizo

Za vzpostavitev vseh treh kanalov z gručo skrbi sprednji del *proizvajalca ClusterActor*, katerega naloge so:

- Pošiljanje zahteve za analizo zadnjemu delu *proizvajalca.*
- Čakanje na prostega *izvajalca*.
- Ko je *izvajalec* na voljo, zaprosi za začetek analize (*AnayserProtocol.Start*), mu sporoči pozicijo, čas trajanja in globino preiskovanja ter stopnjo težanosti.
- V času analize sprejema obvestila *izvajalca* (ResultUCI, BestMove, Finished) in o tem obvešča uporabnika.
- V primeru napake na katerem koli koraku o tem obvesti uporabnika.

Ker kanali delujejo na različne načine, je za dejansko komunikacijo uporabljen vmesni akter, ki zna uporabiti vzpostavljen kanal in komunicirati s *ClusterActor* akterjem. Ti akterji so:

- ***AsyncActor:*** je akter, ki Play okolju sporoči, da bo na zahtevo odgovoril asinhrono »Action.async«. Ko pridobi rezultat analize, o tem obvesti okolje, ki rezultat posreduje uporabniku. V vmesnem obdobju uporabnik čaka na odgovor, ne da bi prejemal vmesne rezultate in bil obveščen o trenutnem stanju.
- ***ChunkActor:*** je akter, ki komunicira s kanalom, ki se vzpostavil ob zahtevi. S tem se omogoča funkcionalnost razbitega odgovora (angl. chuncked response) z uporabao *Comet* komponente, ki podatke zavije v JavaScript klice, ki se nato izvedejo v brskalniku. *ChunkActor* sproti obvešča uporabnika o stanju, tako da le te sproti vpisuje v vzpostavljeni kanal, kar sproži posodobitev evalvacije na strani spletne aplikacije.
- ***WebsocketActor:*** je akter, ki omogoča dvosmerno komunikacijo z uporabnikom. Uporabnik vzpostavi spletno vtičnico (angl. websocket), preko katere pošilja zahteve za analizo pozicije. Akter mu odgovarja z vmesnimi rezultati analize in končnim stanjem, ko se analiza zaključi. Hkrati dovoljuje obdelavo samo ene zahteve. Za vspostavitev spletne vtičnice se uporablja podpora, ki jo nudi komponenta *Websocket.acceptWithActor*, s pomočjo katere povemo okolju, naj pri vzpostavitvi kanala uporabi WebsoketActor.

```
class AsyncActor () extends Actor with ActorLogging
{
  override def receive = {
    case analyseFen:AnalyseFen => ...
  }

  def waitResult (): Actor.Receive = {
    case res : SocketProtocol.Analysis => ...
    case bestMove: SocketProtocol.BestMove => ...
    case err: SocketProtocol.Error => ...
    case SocketProtocol.Done => ...
  }
}

class ChunkActor (out: Concurrent.Channel[String]) extends Actor with ActorLogging
{
  override def receive = {
    case analyseFen:AnalyseFen => ...
  }

  def waitResult (): Actor.Receive = {
    case res: SocketProtocol.Analysis => ...
    case bestMove: SocketProtocol.BestMove => ...
    case SocketProtocol.Done => ...
  }
}

class WebsocketActor (out: ActorRef) extends Actor with ActorLogging{
  override def receive = {
    case msg:String => ...
  }

  def waitResult (): Actor.Receive = {
    case res : SocketProtocol.Analysis => ...
    case bestMove: SocketProtocol.BestMove => ...
    case err: SocketProtocol.Error => ...
    case SocketProtocol.Done => ...
  }
}
```

Izsek 19: Izvedbe akterjev, ki omgočajo različne tipe povezav

Sistem rezultat vrne v JSON zapisu, v obliki {type:msg_type, message:msg}, kjer type nakaže tip sporočila, message pa je samo sporočilo. Možni tipi sporočil so:

- request_analyse: zahteva za novo analizo. Sporočilo vsebuje FEN pozicijo in opcijsko še stopnjo težavnosti.
- response_analyse: rezultat analize. Sporočlio evalvacijo pozicije.
- response_bestmove: Sporočilo vsebuje najboljšo naslednjo potezo (kot rezultat analize).

- response_done: Sporočilo, da je analiza končana.
- request_ok: Povratno sporočilo, da je zahtevek sprejet.
- request_error : Sporočilo o napakis s številko napake.

Spodaj je izvedba protokola za sporazumevanje z uporabnikom *SocketProtocol*, ki vsebuje možne odgovore sistema in v primeru uporabe WebSocket kanala še možne ukaze v smeri od uporabnika k strežniškemu delu aplikacije. Dodan je še primer komunikacije med uporabnikom in strežnikom.

```
object SocketProtocol {

  // SocketMessage - requests and responses
  abstract class SocketMessage (@transient val messageType: String)
  case class AnalyseFen (fen: String, skill: String = "20") extends
SocketMessage(REQUEST_ANALYSE)
  case class Error (errorCode: Int) extends SocketMessage(RESPONSE_ERROR)
  case class Ok () extends SocketMessage (RESPONSE_OK)
  case class Done () extends SocketMessage (RESPONSE_DONE)
  case class BestMove (move: String) extends SocketMessage(RESPONSE_BESTMOVE)
  case class Analysis (depth: Int, nodes:Int, time:Int, evaluations: List[Evaluation]) extends
SocketMessage(RESPONSE_ANALYSIS)
  case class Evaluation (scoreType: String, scoreValue: Int, moves: String )

  // Error codes
  val ERRORCODE_INVALID_STATE = 1
  val ERRORCODE_WORKER_NOT_AVAILABLE = 2

  // Message Types
  val REQUEST_ANALYSE = "request_analyse"
  val RESPONSE_ANALYSIS = "response_analyse"
  val RESPONSE_BESTMOVE = "response_bestmove"
  val RESPONSE_DONE = "response_done"
  val RESPONSE_OK = "response_ok"
  val RESPONSE_ERROR = "response_error"
  val EVAL_TYPE_CP = "cp"
  val EVAL_TYPE_MATE = "mate"

  // Parsing & Formating
  def parseRequest (request: String) : SocketMessage =...

  implicit val analyseFenWrites = Json.writes[AnalyseFen]
  implicit val errorWrites = Json.writes[Error]
  implicit val evaluationWrites = Json.writes[Evaluation]
  implicit val responseWrites = Json.writes[Analysis]
  implicit val bestmoveWrites = Json.writes[BestMove]

  implicit val analyseFenReads = Json.reads[AnalyseFen]
  implicit val errorReads = Json.reads[Error]
  implicit val evaluationReads = Json.reads[Evaluation]
  implicit val responseReads = Json.reads[Analysis]
  implicit val bestmoveReads = Json.reads[BestMove]
}
```

Izsek 20: Izvedba JSON komunikacijskega protokola

```
// request_analyse: zahteva za izvedbo analize
{"type":"request_analyse","message":{"fen":"4rrk1/1b3Bp1/1n3q1p/2p1N3/1p6/7P/PP3PP1/R2QR1
K1 b - - 0 24"}}

// response_analyse: vmesni odgovor z analizo pozicije
// Storitev pošilja več vmesnih odgovorov v času analize
{"type":"response_analyse","message":{"depth":16,"nodes":2644116,"time":3015,"evaluations":[{"sc
oreType":"cp","scoreValue":0,"moves":"f6d7 d2d4 c8b7 b1d2 e5d4 c3d4 c6a5 b3c2 c7c5 b2b3 c5d4
f3d4 e7f6 c1b2 a5c6 d2f3 c6b4 c2b1 a8c8 a2a4 b5a4 a1a4 a6a5
d1e2"},{"scoreType":"cp","scoreValue":0,"moves":"c8b7 d2d4 f6d7 b1d2 e5d4 c3d4 c6a5 b3c2 c7c5
b2b3 c5d4 f3d4 e7f6 c1b2 a5c6 d2f3 c6b4 c2b1 a8c8 a2a4 b5a4 a1a4 a6a5
d1e2"},{"scoreType":"cp","scoreValue":0,"moves":"c6a5 b3c2 c7c5 d2d4 c5d4 c3d4 c8b7 d4d5 d8c7
c2d3 f6d7 b2b4 a5c4 b1c3 c4a5 c1d2 a5c4 d2c1"}]}}

// response_bestmove : Najboljša naslednja poteza
// Z njim storitev ob koncu analize sporoči najboljšo naslednjo potezo
{"type":"response_bestmove","message":{"move":"f6d7"}}

// response_done : Zaključek analize
{"type":"response_done"}
```

Izsek 21: Primer komunikacije med uporabnikom in sistemom

5.4.2 Spletna aplikacija

Uporabniška spletna aplikacija je razvita v okolju AngularJS po principu enostranske aplikacije (angl. single page application), kjer je aplikacija v celoti naložena na eni spletni strani. Za vse tranzicije med podstranmi skrbi JavaScript in AngularJS okolje, ki dinamično generira in nalaga nove zaslonske maske. Dizajn spletne aplikacije je odziven (angl. responsvie), kar pomeni, da se prilagodi velikosti zaslona in zato uporabna tudi na mobilnih napravah ter tablicah.

Slika 15: Postavitev elementov aplikacije na ožjih zaslonih

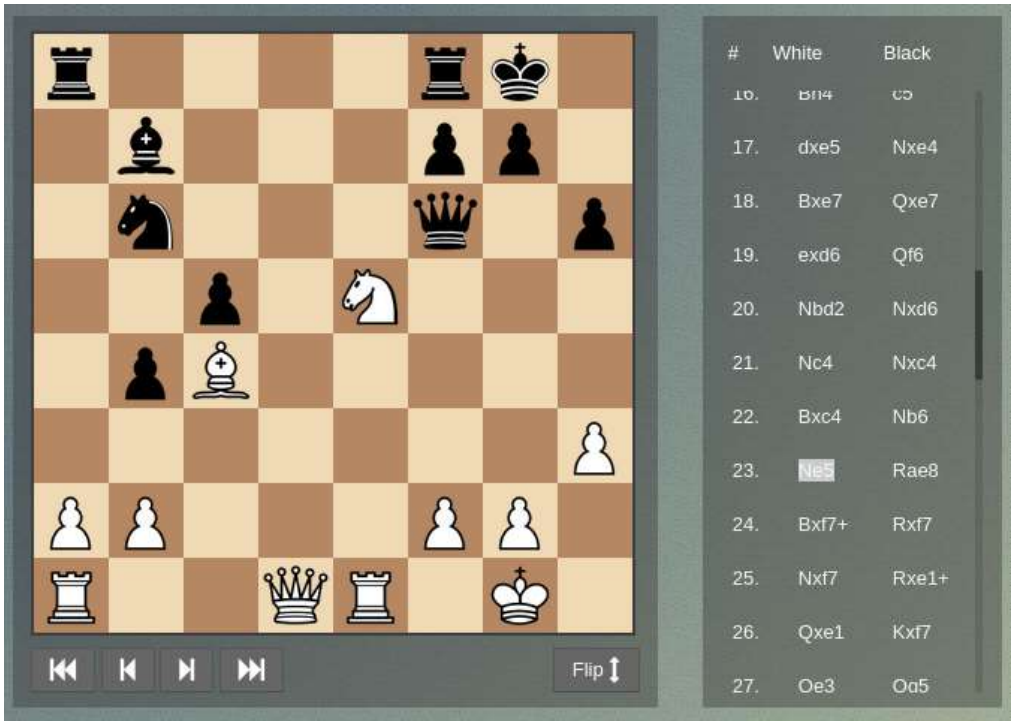
Aplikacija je po AngularJS paradigmi razdeljena na množico krmilnikov, servisov, direktiv in filtrov, ki skupaj omogočajo interakcijo uporabnika z aplikacijo. Glavni deli aplikacije so *vmesnik za analizo šahovskih partij*, *vmesnik za igranje s šahovskim strojem* in *vmesnik za nalaganje iger* v PGN zapisu. Spodaj so navedeni najpomembnejši krmilniki in servisi, ki s pripadajočimi zaslonskimi maskami gradijo uporabniški vmesnik.

Za prikaz šahovkse plošče in vrednotenje pozicij sta uporabljeni knjižici *ChessBoardJS* in *ChessJS,* opisani v poglavju 4.3.5. Združevanje funkcionalnosti knjižic je dosežena s pomočjo naslednjih razvitih komponent:

- Krmilnik ***BoardWidgetContoller*** (in pripadajoča maska) skrbi za interakcijo med uporabnikom in *ChessBoardJS* knjižico, ki omogoči premike med potezami s tipkovnico (tipke naprej in nazaj) in gumbi (naprej, nazaj, začetek, konec). Poleg interakcije je krmilnik zadolžen tudi za spreminjanje velikosti šahovske plošče, in sicer v primeru, da se velikost strani spremeni.
- Krmilnik ***MovesWidgetContoller*** (in pripadajoča maska) je namenjen za prikaz potez v šahovski partiji, ki jo analiziramo. Krmilnik omogoča interakcijo s prikazanimi

potezami, in sicer na način, da lahko uporabnik s klikom na določeno potezo skoči na izbrano pozicijo.

- Servis ***ChessServis*** je vmesnik do *ChessJS* knjižice, s pomočjo katere nadzoruje interno stanje šahovskega poteka in pozicije. Knjižici doda nove funkcionalnosti, ki se uporabljajo v aplikaciji – upravljanje z alternativami, skoki med potezami ter kreiranje šahovske plošče s pomočjo *ChessBoardJS* knjižice. Spodaj je prikazan izsek izvedbe servisa.
- Servis ***KeyBoardServis*** omogoča globalno lovljenje pritiskov na tipkovnico. Krmilnikom ponudi možnost, da se prijavijo na obveščanje, ko uporabnik pritisne določeno tipko. To je omogočeno preko registrirane direktive *keytrap*, ki se lahko doda poljubnemu elementu (v aplikaciji je to kar koren). To ob nalaganju strani povzroči, da direktiva registrira poslušalca za pritiske tipk na izbrani komponenti oz. sestavini in o tem obvešča poslušalce.



Slika 16: Šahovska plošča in navigacija

```
define(['angular', 'chessboardjs', 'chessjs'],function() {

function ChessService(){

  // ... Initialization ...

  // ... Support functions (getters, setters)

  // ChessBoard functions
  service.createBoard = function () { ... };

  // ChessJS functions
  service.createGame = function () {...}
  service.createGamePGN = function (pgn) {...}
  service.createGameUCI = function (fen, moves) {...}
  service.initGameData = function (game) {...}

  // ChessBoard <-> ChessJS integration
  service.goStart = function () {...}
  service.goBack = function() {...}
  service.goNext = function() {...}
  service.goEnd = function() { ... }
  service.moveToPath = function (path, ply) { ... }
}

return ChessService;
 );
```

Izsek 22: Izvedba servisa *ChessService*

Za pošiljanje zahtev za novo analizo skrbi servis ***AnalyseService*****,** ki pošilja zahtevke preko spletne vtičnice (angl. websocket) in razdeljenega odziva (angl. chunked response), s katerim je dosežena združljivost s starejšimi brskalniki. Servis je zadolžen za sprejemanje zahtevkov s strani krmilnikov za anlizo kot tudi krmilnikov, namenjnih igri s šahovskim strojem**.** Servis poskrbi za razčlenjevanje dobljenih podatkov in obveščanje krmilnikov o novih evalvaciji. Spodaj je prikazan izsek izvedbe servisa.

```
define(['angular', 'chessboardjs', 'chessjs'],function() {

function AnalyseService($rootScope, ChessService){

    // ... Initialization ...

    // Request analyse (comet or websocket)
    service.requestAnalyse = function(_game, _skill,  _callback) {...};

    // Responses in JSON
    service.update = function (data) { ... };
    var processBestmove = function (bestmove) { ... };
    var processResult = function (res) { ... };
    var processDone = function () { ... };
    var processError = function (err) { ... };


    // Comet implementation
    var requestAnalyseComet = function (fen, skill) { ... };

    // WebSocket implementation
    var initSocket = function () { ... };

    return service;
}

return AnalyseService;
});
```

Izsek 23: Izvedba servisa *AnalyseService*

Za pošiljanje zahtevkov servisu in prikaz evalvacij skrbi ***AnalyseWidgetController***, ki sproti obvešča uporabnika o poteku analize. Po končani analizi pa uporabniku omogoči pregled vseh dobljenih šahovskih poti.

Slika 17: Uporabniški vmesnik za prikaz analiz šahovske pozicije

Krmilnik ***PlayController*** je zadolžen za zmožnost igranja igre s šahovskim strojem. S pomočjo vmesnika omogoča nastavljanje težavnostne stopnje in izbiro barve. Omogoča tudi izbiro začetne pozicije, katere funkcionalnost je uporabljena s strani komopnente za analiziranje, ki omogoči uporabniku igranje trenutno prikazane pozicije. Funkcionalnost igranja s šahovskim strojem deluje po principu, da se za naslednjo potezo stroja zahteva analiza trenutne pozicije (z vključeno težavnostno stopnjo), katere rezultat je poteza, ki jo odigra aplikacija.

Slika 18: Uporabniški vmesnik za igranje s šahovskim strojem

Spodaj je izsek izvedbe krmilnika *PlayController,* ki skrbi za kreiranje nove igre, sledenje premikov in pravilnost le-teh, zahtevanje analize ter posodabljanje plošče in statusa.

```
define(['chessjs', 'chessboardjs'], function() {
function PlayController($scope, $modal, ChessService, AnalyseService) {

 // ... Initialization ...

 // Analysis Callback
 var callback = {};
 callback.onBestmove = function(move) { ... };
 callback.onError = function(error) { ... };
 var playmove = function (move) { ... };

 // Request analyse (based on user interactions)
 var analyse = function() { ... };
 var onDragStart = function(source, piece, position, orient) { .. };
 var onDrop = function(source, target) { ... };
 var onSnapEnd = function() { ... };

 // Update board and status
 var update = function () { ... };
 var updateStatus = function() { ... };

 // Initialize board
 var cfg = { /* Board configuration */ };
 $scope.board = new ChessBoard2('board', cfg);
 ChessService.setCurrentBoard($scope.board);
 $scope.createNewGame = function (color, strength) { };


 // New-Game modal handler
 $scope.newgame = function () {
   var modalInstance = $modal.open({
     templateUrl: 'assets/html/modals/modal-newgame.html',
     controller: ModalInstanceCtrl
   });

   modalInstance.result.then(
       function (res) { /*Create new game */ },
       function ()    { /* Dismiss dialog */ });
 };
}

// New-game modal
var ModalInstanceCtrl = function ($scope, $modalInstance, ChessService) {
  $scope.computer = { /* Initial settings */ };
  $scope.getStrenghts = function () { ... };
  $scope.play = function () { ... };
  $scope.cancel = function () { ... };
};

PlayController.$inject = [];
return PlayController;

});
```

Izsek 24: Izvedba krmilnika *PlayController*

Za vnos igre, namenjene analizi, skrbi krmilnik ***PgnLoadController***, ki preveri veljavnost vnosa in preusmeri uporabnika na uporabniški vmesnik namenjen analizi. V primeru napake

pri preverjanju verljavnosti vnosa o tem obvesti uporbnika. Podprt je PGN zapis. Sodaj sta prikazana uporabniški vmesnik in izsek izvedbe krmilnika.

Slika 19: Uporabniški vmesnik za vnos iger v PGN zapisu

```
define(function() {

function PgnLoadController($scope, $modal, $log, $location, ChessService) {

   // ... initialization ...

   // Open PGN load modal dialog
        $scope.open = function () {

                var modalInstance = $modal.open({
                        templateUrl: 'partials/modal-loadpgn.html',
                        controller: ModalInstanceCtrl
                });

                modalInstance.result.then(
                   function (game) { /*  load game (ChessService) */ },
                   function ()     { /* show error dialog */}
                );
        };
}

var ModalInstanceCtrl = function ($scope, $modalInstance, ChessService) {
        $scope.alerts = [];
        $scope.analyse = function (pgn) { /* Parse pgn */};
        $scope.cancel = function () { ... };
        $scope.closeAlert = function(index) { ... };
};

PgnLoadController.$inject = [];
return PgnLoadController;
});
```

Izsek 25: Izvedba krmilnika *PgnLoadController*

Poglavje 6 Postavitev in upravljanje

Poglavje v prvem delu opisuje postavitev storitve ChessCloud v oblak in predstavi razvitih orodij, ki so pri tem uporabljena. Za hitrejše in preprostejše vzpostaljanje infrastrukture v oblaku, postavljanje servisa in upravljanje z njim so razvite Python skripte, ki skupaj z uporabo Fabirc in CloudLib, izpostavijo priročen in širok nabor ukazov, ki so izvršljivi iz ukazne lupine.

Ker je potrebno po postavitvi storitev spremljati, je v drugem delu poglavja opisan sistem za monitoriranje ter preprost sistem za obremenitveno testiranje, s katerim je moč preveriti delovanje storitve. V ta namen je bil razvit nastavljiv Gattling scenarij, ki posnema konstanten in naraščajoč obisk ChessCloud spletne storitve.

6.1 Postavitev sistema

Storitev ChessCloud je grajena na javanskih tehnologijah, kar bi omogočalo uporabo oblaka s storitvenim modelom platforma kot storitev (PaaS), vendar ker je sestavni del storitve tudi šahovski stroj (tj. StockFish), ki mora biti nameščen na *izvajalskih* vozliščih, je potrebna kontrola tudi nad infrastrukturo, kar implicira uporabo oblakov ki ponujajo infrastrukturo (IaaS).

Celotni postopek postavitve je avtomatiziran. Skripte omogočajo postavitev rešitve ali v zasebni OpenStack oblak ali v javni AWS EC2 oblak. Uporabljen operacijski sistem je Linux Debian, za katerega se, pripravijo instalacijski paketi (deb paketi). Nad postavljeno storitvijo pa so avtomatizirana tudi najbolj pogosta opravila: ustavitev in zagon analitične gruče in ostalih komponent, dodajanje in odvzemanje vozlišč (angl. scale-out in scale-in), izvajanje linux komand ali ukazov, pregled dnevniških datotek, itd.

Skripte so napisane v programskem jeziku Python, z uporabo knjižice *Fabric* in *CloudLib*, opisani v poglavju 4.2. Kombinacija obeh omogoča upravljanje s poljubno oblačno infrastrukturo in oddaljenimi viri preko ukazne lupine. Vmesnik je sestavljen iz ukazov, ki delujejo globalno na celotni sistem (npr. uvodna nastavitev storitvene infrastrukture,

zaustavitev sistema, itd.), in iz ukazov, ki delujejo na izbrana vozlišča. Spodaj je naveden primer za vsak tip.

- $>fab create_cluster : globalna akcija, ki povzroči uvodno nastavitev storitvene infrastrukture v izbranem oblaku.
- $>fab workers stop : akcija, ki zausatvi vsa izvajalska vozlišča v sistemu. V tem primeru prvi del ukaza pove, nad katerimi vozlišči bo akcija izvršena.

Rešitev je razdeljena na dve osnovni komponenti: *infrastrukturna komponenta* in *storitvena komponenta*, ki sta nastavljivi preko konfiguracijske datoteke (pravice dostopa, izbira servisov, itd.)

6.1.1 Infrastruktura

Infrastrukturna komponenta omogoča povezavo na oblačne storitve, postavitev oziroma uvodno nastavitev vozlišč, iskanje le-teh glede na vlogo v gruči ter upravljanje z njimi. Infrastrukturna komponenta je odvisna od gonilnikov, ki abstrahirajo inicializacijo povezave s ponudniki.

Gonilniki (drivers.py) zagotavljajo abstrakcijo inicializacije CloudLib gonilnikov. Namen modula je, da gonilnik s pripadajočimi nastavitvami zavije in tako doseže neodvisnost od nadaljnjih korakov. Za to je zadolžen *LibCloudWrapper*, ki poleg inicializcije gonilnika zagotavlja tudi dostop do tipov uporabljenih instanc, privzetih slik, itd., ki jih infrastrukturni modul uporabi pri gradnji infrastrukture. Podprta sta zasebni OpenStack in javni AWS EC2 oblak. CloudLib podpira še celo vrsto ostalih oblakov, tako da je dodajanje podpore, da storitev teče tudi na ostalih oblakih, dokaj preprosta.

Infrastrukturna komponenta (infrastructure.py) zagotavlja akcije, namenjene upravljanju z infrastrukturo, izvajanje osnovnih akcij nad vozlišči in pridobivanje trenutnega stanja sistema. Na voljo so naslednji ukazi:

- Ukazi, namenjeni gradnji sistema in upravljanje z njim:
  - $>fab [create_cluster|destroy_cluster|reboot_cluster] : ukazi namenjeni postavitvi in uničevanju celotene infrastrukture ter ponovno zaganjanje vsej vozlišč. Ukaz create_cluster omogoča preko argumentov definiranje velikosti gruče; število vozlišč posameznega tipa.
  - $>fab create_<role>: kreiranje novega vozlišča glede na vlogo: frontend, master, worker.
  - $>fab <selection> destroy: uničevanje izbranih vozlišč.

- Izborni ukazi (<selection>) izberejo vozlišča, nad katerimi se bo izvršila akcija. V primeru, da akcija ni podana, se ukaz uporablja za izpis vozlišč z imeni in njihovimi internetnimi naslovi:
    - $>fab [all|cluster] <action>: ukaz za izvajanje akcije ali nad vsemi vozlišči v sistemu ali vsemi vozlišči, ki sestavljajo gručo.
    - $>fab [masters|workers|frontends] <action> : ukaz za izvajanje akcije nad vozlišči določenega tipa (masters, workers, frontends, cluster, all).
    - $>fab select <action>: ukaz, ki izvede akcijo nad izbranim vozliščem. Sistem v prvem koraku izpiše vsa vozlišča; uporabnik izbere enega.
- Ukaz, namenjen uvodni nastavitvi računalniških virov (angl. provisioning):
    - $>fab <selection> provision: poskrbi za nastavitve računalniškega vira in inštalacijo programskih paketov, potrebnih za zaganjanje ChessCloud storitve.
- Ostali ukazi, namenjeni osnovnemu manipuliranju z vozlišči:
    - $>fab ssh: ukaz za vzpostavitev ssh povezave do poljubnega vozlišča. Izpiše vsa vozlišča z zaporedno številko, uporabnik pa izbere eno vozlišče.
    - $>fab <selection> cmd=<cmd>: ukaz za izvajanje komande oziroma ukaza <cmd> na izbranih vozliščih.

Spodaj je prikazanih nekaj izvedb ukazov. Prvi izsek prikazuje ukaze za postavitev in uničevanje gruče. Drugi izsek je izbira vozlišč glede na tip, ki ga v nadaljevanju združimo z željenim ukazom, ki se izvede na vsakem izmed izbranih vozlišč. Tretji izsek prikazuje rutino, ki omogoča izvajanje ukaza nad poljubnim vozliščem, ki ga uporabnik izbere izmed vsemi, ki so na voljo.

```
@task
def create_cluster (frontends=1, masters=1, workers=2):
        print 'Creating '+CLUSTER_NAME_PREFIX+' cluster'
        create_masters(int(masters))
        create_workers(int(workers))
        create_frontends(int(frontends))
        wait_for_running_state()
@task
def destroy_cluster ():
        print 'Destroying '+CLUSTER_NAME_PREFIX+' cluster'

        cluster_nodes = get_nodes(prefix=CLUSTER_NAME_PREFIX)

        for node in cluster_nodes:
                print 'Destroying %s...' % node.name
                node.destroy()

def create_masters(size_masters):
        image = get_image(wrapper.get_property('DEFAULT_IMAGE'))
        size = get_size(wrapper.get_property('MASTER_SIZE_NAME'))
        for idx in range (0,size_masters):
                create_node(MASTER_NAME_PREFIX+`idx+1`, image, size)

def create_workers(size_workers):
        for idx in range (0,size_workers):
                create_worker(WORKER_NAME_PREFIX+`idx+1`)

def create_worker(name):
        image = get_image(wrapper.get_property('DEFAULT_IMAGE'))
        size = get_size(wrapper.get_property('WORKER_SIZE_NAME'))
        create_node(name, image, size)

def create_frontends(size_frontend):
        image = get_image(wrapper.get_property('DEFAULT_IMAGE'))
        size = get_size(wrapper.get_property('FRONTEND_SIZE_NAME'))
        for idx in range (0,size_frontend):
                create_node(FRONTEND_NAME_PREFIX+`idx+1`, image, size)

def create_node (name, image, size):
        print "Creating node >> [%s , %s , %s]" % (name, image, size)
        res = driver.create_node(name=name, image=image, size=size,
                ex_keyname=wrapper.get_property('ex_keyname'))
```

Izsek 26: Izvedbe globalnih ukazov za kreiranje in uničevanje gruče

```
@task
def masters():
        env.role='master'
        env_hosts(MASTER_NAME_PREFIX)

@task
def workers():
        env.role='worker'
        env_hosts(WORKER_NAME_PREFIX)

@task
def frontends():
        env.role='frontend'
        env_hosts(FRONTEND_NAME_PREFIX)

def env_hosts (prefix):
        cluster_nodes = get_nodes(prefix='')
        env.seeds=[]
        env.hosts=[]
        env.host_names={}
        env.host_roles={}

        print prefix

        for node in cluster_nodes:
                if (node.state != NodeState.RUNNING):
                        continue
                if node.name.startswith (prefix):
                        print "[*] %-22s| %s | %s " % (node.name,
                   node.private_ips, node.public_ips)
                        node_ip = get_node_ip(node)
                        env.hosts += [node_ip]
                        env.host_names[node_ip] = node.name
                        env.host_roles[node_ip] = get_node_role(node)
```

Izsek 27: Izvedba izbirnih ukazov za vse tipe vozlišč

```
@task
def select():

        if env.hosts is None or len(env.hosts) == 0:
                env_hosts('')

        idx = 0
        for host in env.hosts:
                idx += 1
                print "[%s] %-22s| %s " % (idx, env.host_names[host], host)

        if idx > 1:
                host_idx = int(prompt('Please chose host: '))
        else:
                host_idx = 1

        host = env.hosts[host_idx-1]
        print host

        env.host = host
        env.hosts = [host]
        env.host_string = host
```

Izsek 28: Izvedba izbirnega ukaza za poljubno vozlišče

6.1.2 Storitev

Storitvena komponenta omogoča postavitev storitve na postavljeno infrastrukturo skupaj z nastavitvami gruče, pripravo namestitvenih paketov ter upravljanje s samo storitvijo.

Namestitveni paket je pripravljen za operacijski sistem Linux Debian. Paket omogoča preprosto nameščanje in odstranjevanje storitve (dpkg) ter njeno zaganjanje in ustavljanje; service chesscloud-<role> [stop|start|restart]. Pri namestitvi se registra upstart servis, ki zažene aplikacijo ob nalaganju operacijskega sistema ter jo ob zaustavitvi tudi pravilno ugasne. Poleg tega skrbi, da se aplikacija v primeru nepričakovanega sesutja ponovno zažene. Za varno izvajanje storitve se kreirata tudi nov linux uporabnik in grupa (chesscloud-<role>), ki se uporabi za zaganjanje storitve z okrnjenimi pravicami.

Pri pripravi namestitvenega paketa, namestitvene skripte nastavijo semena (angl. seed nodes), ki jih storitev uporabi za povezovanje vozlišč v gručo. To je mogoče avtomatizirati, ker lahko storitvena komponenta dostopa do postavljenih vozlišč in njihovih internetnih naslovov. Za semenska vozlišča se uporabijo kar vozlišča tipa master.

Storitvena komponenta (service.py) zagotavlja akcije, namenjene upravljanju in nameščanju ChessCloud storitve ter pridobivanje njenega stanja. Za dostop do infrastrukture in vozlišč uporablja funkcionalnosti infrastrukturne komponente.

- Ukazi, namenjeni nameščanju sistema:
    - $>fab package=<role> : ukaz prevede izbrano rešitev, pripravi namestitveni paket in ga nastavi tako, da se rešitev zna povezati v gručo.
    - $>fab <selection> [deploy|remove]: ukaz deploy na izbrana vozlišča prenese produkcijski paket na oddaljen računalniški vir in ga tam namesti. Ukaz remove poskbi za pravilno odstranjevanje rešitve.
    - $>fab <selection> rollback: ukaz iz selekcije vozlišč odstrani trenutno verzijo in namesti starejšo, ki si jo lokalno shrani pri namestitvi.
    - $>fab deploy_<role> : ukaz poskbrbi za oba koraka hkrati; pakiranje in nameščanje. Pri nameščanju poskrbi za zaustavitev prejšnje verzije in zagon novo nameščene.
- Ukazi, namenjeni zaganjanju in zaustavljanju sistema, za razliko od infrastrukturne komponente, ki deluje nad računalniškimi viri, delujejo nad nameščeno storitvijo:
    - fab [start_cluster|stop_cluster] : ukaza, namenjena zaganjanju in zaustavljanju celotnega sistema.

- fab <selection> [start|stop]: ukaza, namenjena zaganjanju in zaustavljanju komponent, nameščenih na izbranih vozliščih.

- Ukazi, namenjeni horizontalnemu skaliranju sistema:
  - fab <role> scale_out : ukaz pripravi novo vozlišče izbranega tipa, počaka, da se inicializira in postane dostopen, nastavi sistem (angl. provisioning) ter na koncu namesti storitev (deploy).
  - fab <role> scale_in : ukaz odstrani (angl. terminate) vozlišče izbranega tipa. V primeru, da je vozliče eno samo, se ukaz ne izvede.

- Ukazi, namenjeni pridobivanju stanja:
  - fab select [tail_log|get_log|clean_log] : ukaz za pregled, pridovbivanje in brisanje log datotek iz izbranih vozlišč.

Spodaj so prikazani primeri izvedb ukazov. Prvi izsek prikazuje izvedbo ukaza package, ki pripravi in nastavi paket. Drugi izsek prikazuje ukaze, ki ta paket na vozlišču nastavijo oziroma odstranijo. Tretji izsek pa prikazuje, kako je s pomočjo skript omogočeno horizontalno skaliranje sistema.

```
def package(role, compile='yes'):

        # ... Variables initialization based on “role” ...

        try:
                ### PREPARE ###
                SED_CMD = 'sed -i s/%s/%s/g %s'
                local ("cp %s %s" % (seeds_conf, seeds_conf+".orig"))
                local (SED_CMD % (SEEDS_PLACEHOLDER, compose(seeds()), seeds_conf))

                if (role != "frontend"):
                        local ("cp %s %s" % (application_ini, application_ini+".orig"))
                        local (SED_CMD % (CLUSTER_ROLE_PLACEHOLDER, role, application_ini))


                ### BUILD ###
                local ('cd %s && exec sbt debian:packageBin' % app_location)

                local ('mkdir -p %s' % LOCAL_FOLDER_DIST)
                local ('cp %s/target/chesscloud-*.deb %s/%s' %
                        (app_location, LOCAL_FOLDER_DIST, env.debfile))

        finally:
                ### CLENAUP ###
                local ("mv %s %s" % (seeds_conf+".orig", seeds_conf))
                if (role != "frontend"):
                        local ("mv %s %s" % (application_ini+".orig", application_ini))
```

Izsek 29: Izvedba ukaza package

```
@task
@parallel
def install():
        role = env.host_roles[env.host]
        env.debfile = "chesscloud-cluster-%s.deb" % role
        put (("%s/%s" % (LOCAL_FOLDER_DIST, env.debfile)), (".."))
        sudo ("dpkg -i --force-all %(debfile)s" % env)

@task
@parallel
def remove():
        role = env.host_roles[env.host]
        env.service = remote_service()
        sudo ("dpkg -r --force-all %(service)s" % env)

@task
@parallel
def rollback():
        remove()
        role = env.host_roles[env.host]
        env.debfile = "chesscloud-cluster-%s.deb" % role
        run ('mv %(debfile)s %(debfile)s_temp' % env)
        run ('mv %(debfile)s_old %(debfile)s' % env)
        run ('mv %(debfile)s_temp %(debfile)s_old' % env)
        install()
```

Izsek 30: Izvedba ukazov za nameščanje in odstranjevanje storitve

```
@task
def scale_out (role="worker"):
        name = get_unique_node_name(role)
        print 'Creating ' + name + '\n'
        create_node(role, name)
        print '\nWaiting for (running state)' + name
        wait_for_running_state()
        node = get_nodes (name, refetch=True)[0]
        env_host(node)
        print '\nWaiting for (ssh) ' + name + '\n'
        execute(wait_for_ssh)

        print '\nProvisioning ' + name + '\n'
        execute(provision)
        print '\nDeploying ' + name + '\n'
        execute(deploy)

@task
def scale_in(role="worker"):
        worker_nodes = get_nodes(prefix=WORKER_NAME_PREFIX)

        if len(worker_nodes) > 1:
                print 'Scaling down ...'
                node = worker_nodes[-1]
                env_host(node)
                print '\nStoping %s ... \n' % node.name
                execute(stop)
                print '\nDestroying %s ...\n' % node.name
                node.destroy()
```

Izsek 31: Izvedba ukazov, ki omgočata horizontalno skaliranje vozlišč

6.1.3 Nastavitve

Sistem avtomatizacije je nastavljiv s pomočjo nastavitvene datoteke *(conf.ini)*. Nastavitve so razdeljene na splošne (globalne) in nastavitve odvisne od ponudnika infrastrukture (specifične). Splošne zajemajo osnovne nastavitve, ki pa jih lahko nastavitve posameznega ponudnika prepišejo:

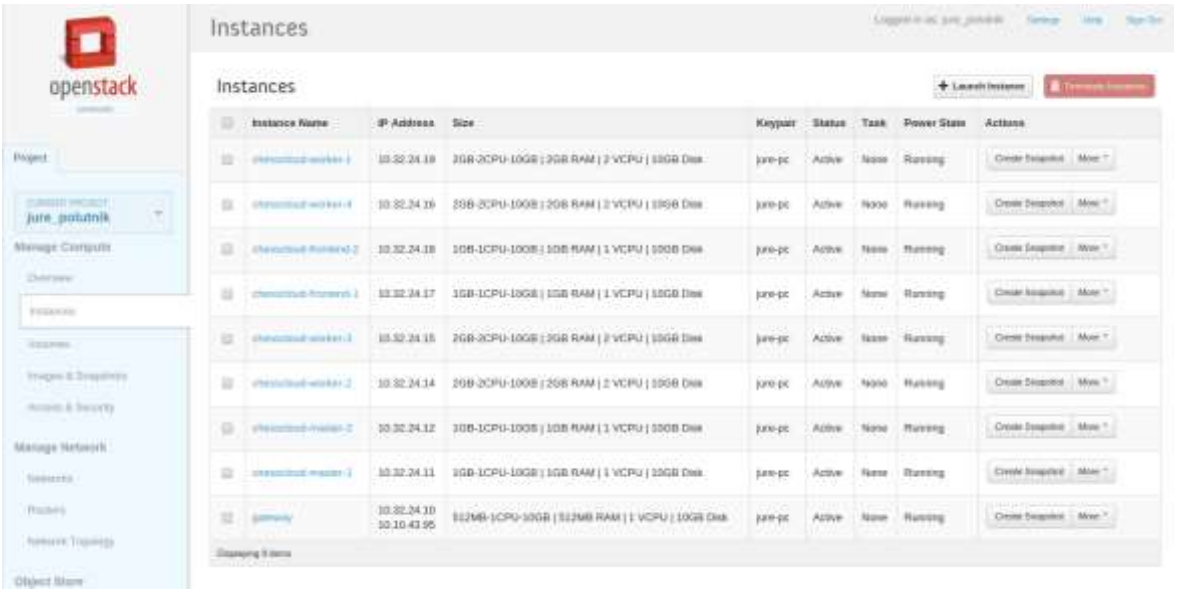
- Pravice dostopa do vozlišč (uporabniško ime, geslo, kjuč).
- Zapis poimenovanja vozlišč glede na vlogo (npr. chesscloud-master-<id>).
- Ime slike operacijskega sistema za inicializacijo infrastrukture.
- Izbira gonilnika (OpenStack ali AWS).

Nastavitve posameznih ponudnikov zajemajo tipe navideznih strojev, ki se bodo uporabili za določen tip vozlišča, ter nastavitve dostopa do posamezne storitve (npr. AWS potrebuje identifikacijo uporabnika in dostopni skriti ključ, medtem ko OpenStack potrebuje tudi naslov kjer se le-ta nahaja, ker gre za zasebno infrastrukturo).

6.2 Chess-Cloud.com

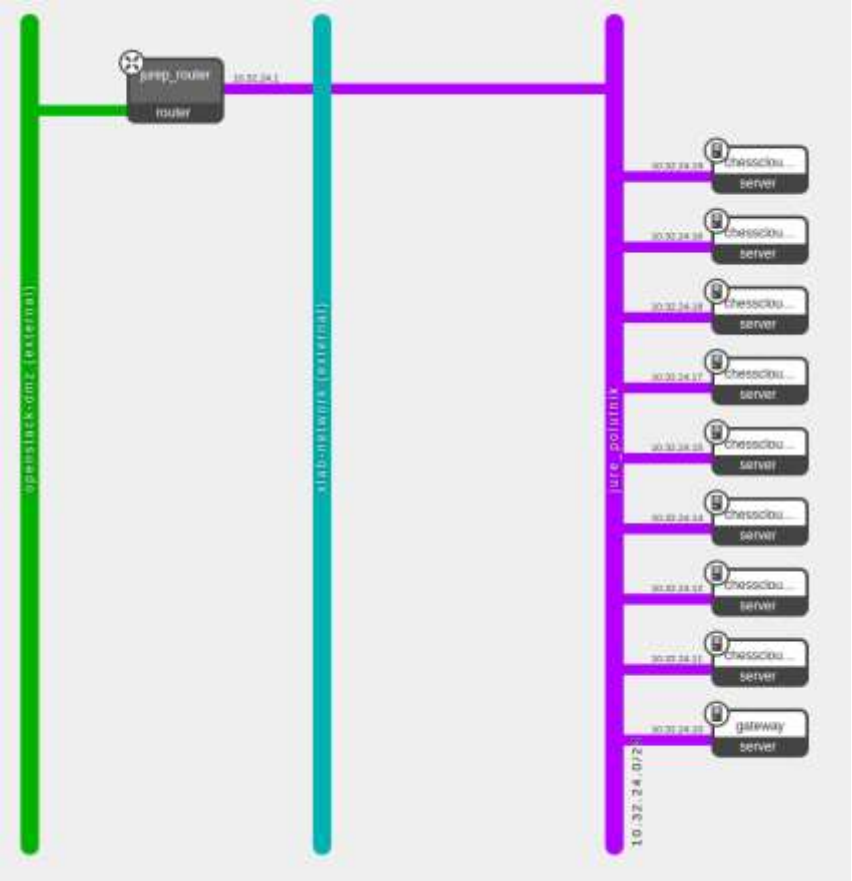
Storitev ChessCloud je dostopna javno na domeni ***chess-cloud.com***. Storitev je postavljena na privatnem OpenStack oblaku in je sestavljena iz:

- 2 Frontend vozlišča; 1CPU (2,5GHz) in 1GB RAM
- 2 Master vozlišča; 1CPU (2,5GHz) in 1GB RAM
- 4 Worker vozlišča; 2CPU (2,5GHz) in 2GB RAM
- Gateway; 1CPU (2,5GHz) in 512MB RAM



Slika 20: ChessCloud vozlišča (OpenStack Dashboard)

Slika prikazuje seznam vseh vozlišč ChessCloud storitve (začetno stanje, saj se vozlišča lahko poljubno dodajajo in odvzemajo). Kot je razvidno iz slike, imajo vsa vozlišča ChessCloud sistema samo zasebni ip in so dostopna samo znotraj svoje virtualne podmreže (10.32.24.0/24). Tako je edina vstopna točka v sistem vozlišče *gateway*, kateremu je dodeljen *plavajoči ip,* s pomočjo katerega vozlišče vidno znotraj interne mreže podjetja. Posredovanje poizvedb na domeno *chess-cloud.com* do *gateway* vozlišča je omogočeno z uporabo glavnega postredniškega strežnika. Na *gateway* vozlišču je nastavljen Nginx, ki služi kot porazdeljevalec obremenitve (angl. load-balancer), ki po principu *round-robin* razvrščevalnega algoritma deli poizvedbe med registrirana *frontend* vozlišča.

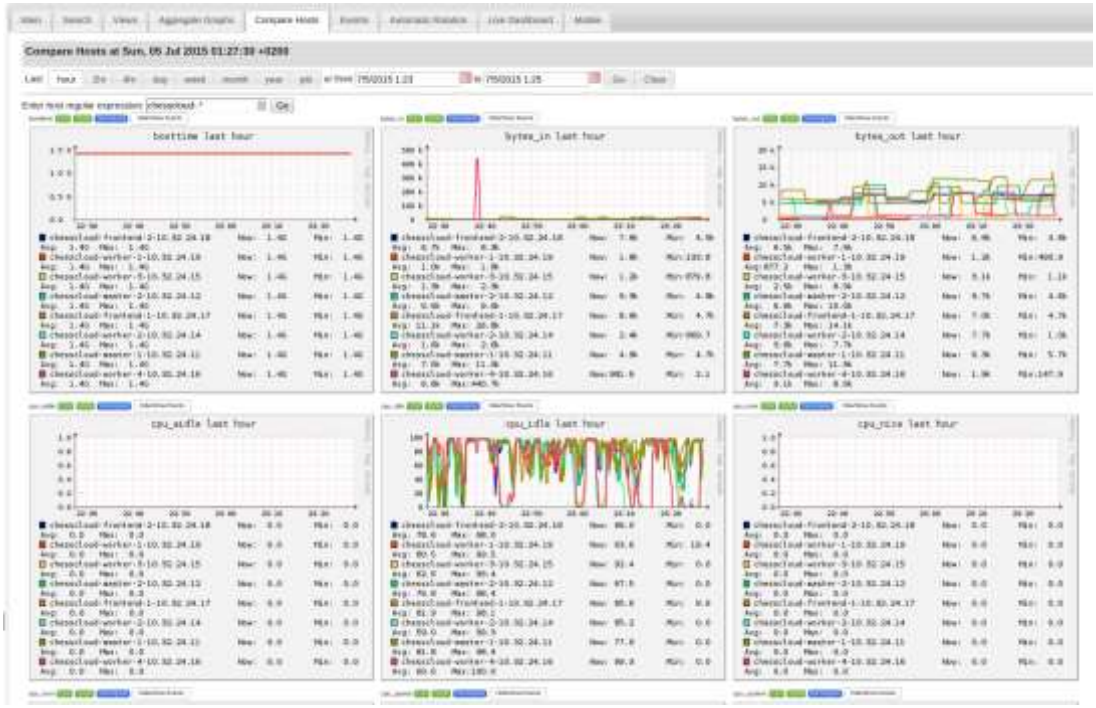


Slika 21: Omrežna topologija (OpenStack)

6.2.1 Spremljanje delovanja sistema

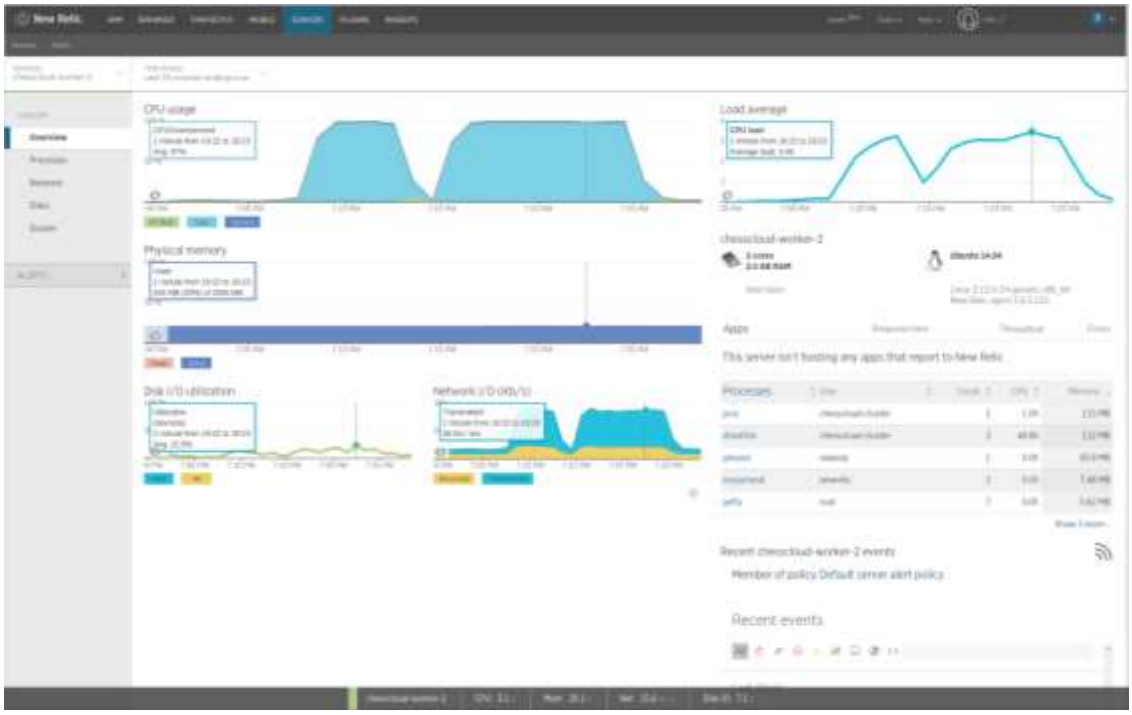
Za spremljanje delovanja sistema je uporabljena odprtokodna rešitev Galnglia, poleg nje pa napredna komercialna rešitev NewRelic, ki zastonj ponuja okrnjeno različico za manjše projekte.

Ganglia je podprta na nivoju privatnega OpenStack oblaka in je namenjena osnovnemu spremljanju infrastrukture. Omogoča spremljanje stanja celotnega sistema v poljubnem časovnem obdobju, preko posameznih in združenih pogledov. Spremljajo se podatki o uporabi procesorske moči, stanju podatkovnih enot, stanju pomnilnikov in uporabi omrežja. Glavna prednost pred NewRelic rešitvijo je, da je rešitev v domeni privatne ali zasebne infrastrukture in omogoča preglede poljubnih časovnih obdobij ter da je frekvenca zapisovanja podatkov večja. Slednja slika priazuje pregled vseh izbranih meritev na vseh vozliščih gruče v obdobju zadnje ure.



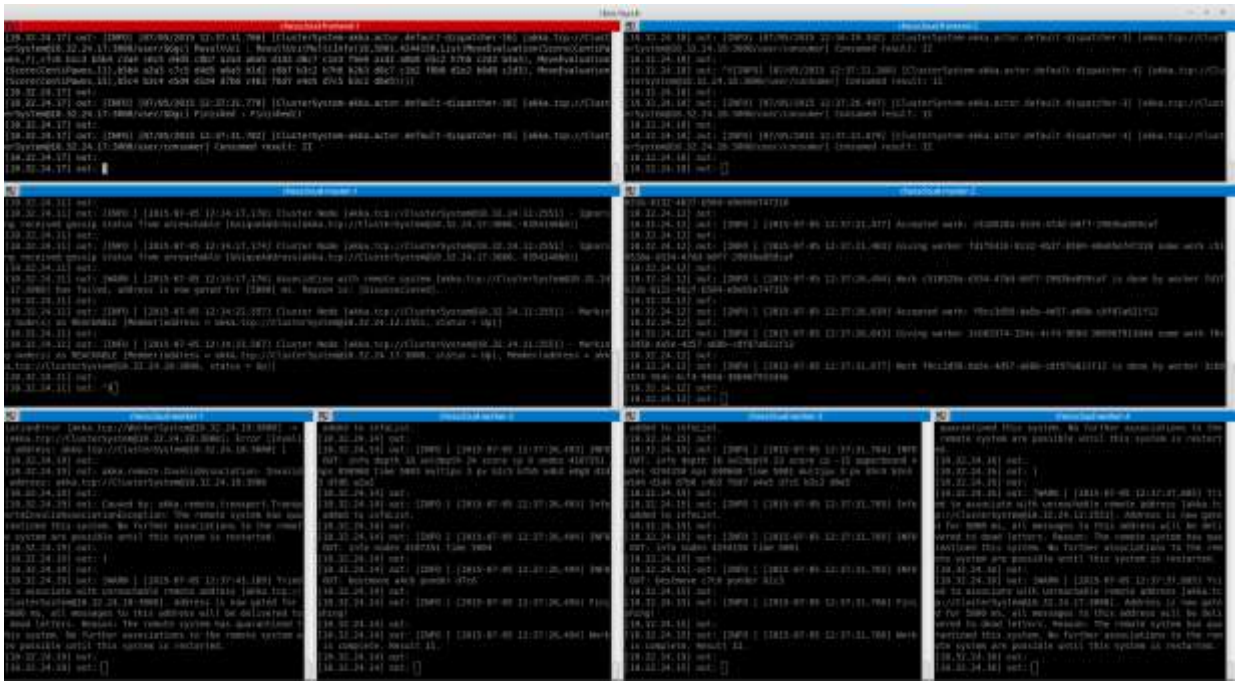
Slika 22: ChessCloud gruča v sistemu Ganglia

Kot dodatna storitev za spremljanje se uporablja NewRelic, ki je ena izmed SaaS storitev, ki omogočajo spremljanje poljubnih sistemov in aplikacij. Za namen ChessCloud storitve je uporabljena zmožnost nadzora nad strežniško infrastrukturo in nadzor nad nameščeno aplikacijo; spletna aplikacija in gruča. Infrastrukturni del je podoben Ganglii z razliko, da omogoča nastavljanje obveščanja o posebnih dogodkih (npr. pomanjkanje prostora, pomnilnika ali procesorske moči). Nadzor nad aplikacijo AMP (Application Performance Montitoring) pa omogoča podrobno spremljanje dogajanja znotraj aplikacije in okolja, nad katerim teče (v primeru ChessCloud storitve je to javanski virtualni stroj). AMP omogoča tudi preverjanje odzivnosti aplikacije in njeno dostopnosti, analizo skalabilnosti sistema in preverjanje SLA (Service Level Agreement). V zastonjski različici je kar nekaj funkcionalnosti onemogočenih in se APM uporablja predvsem za spremljanje javanskega virtualnega stroja, nadzor nad transakcijami in preverjanje dostopnosti aplikacije in gruče ter obveščanje o morebitnih problemih.



Slika 23: ChessCloud vozlišče v sistemu NewRelic

Poleg samega spremljanje aplikacij in serverjev je pri razvoju in iskanju napak še vedno potrebno pregledovati dnevniške datoteke. Slika prikazuje uporabo tail_log ukaza storitvene komponente, ki izvede ukaz tail –f <log_file> na poljubnem vozlišču in tako omogoča spremljanje dnevniških datotek v realnem času. Po potrebi je mogoče prenesti vse dnevniške datoteke z uporabo get_log ukaza. Obstaja kar nekaj sistemov, ki omogočajo osrednji pogled na dogodke in obdelavo le-teh, vendar je za potrebe ChessCloud storitve trenutna rešitev popolnoma zadostna.



Slika 24: Prikaz dnevniških zapisov za celotno gručo (v realnem času)

6.2.2 Obremenitveno testiranje sistema

Za preverjanje delovanja sistema in iskanje kapacitete sistema je uporabljeno orodje Gattling, ki omogoča pripravo dinamičnih scenarijev, s katerimi obremenimo ciljno spletno storitev. Za potrebe ChessCloud storitve je sta pripravljena dva preprosta scenarija, kjer navidezni uporabniki neprestano zahtevajo analizo šahovskih pozicij:

- **stabilen**: stalno število navideznih uporabnikov
    - nastavitve: število navideznih uporabnikov in dolžina testiranja
- **naraščajoč** : naraščajoče število navideznih uporabnikov
    - nastavitve: število navideznih uporabnikov na začetku in koncu, hitrost dodajanja novih in dolžina testiranja

Spodaj je izsek, ki prikazuje izvedbo Gattling simulacije, ki omogoča oba scenarija. Simulacija je nastavljiva s pomočjo nastavitvene datoteke (load-test.conf), ki omogoča nastavljanje lokacije servisa, željene pozicije v FEN zapisu, tip scenarija, njegovo dolžino (in dolžino naraščanja) ter željeno število navideznih uporabnikov.

```
class BasicSimulation extends Simulation {

  val conf = ConfigFactory.load("load-test")

  // Scenario configuration
  val hostname = conf.getString("server.hostname")
  val fen = conf.getString("test.fen") replaceAll (" ", "%20")
  val httpConf = http.baseURL(hostname)

  val scn = scenario("ChessCloud Analysis Scenario").forever() {
    exec(http("request_analyse").get("/analyse/"+fen).
           check(regex("response_analyse").exists))
  }

  // Run configuraiton
  val scenarioType = conf.getString("server.hostname")
  val users = conf.getInt("test.users")
  val rampDuration = conf.getInt("test.rampDuration")
  val duration = conf.getInt("test.duration")

  scenarioType match {
    case "linear" =>
      setUp(scn.inject(atOnceUsers(users)).
        protocols(httpConf)).maxDuration(duration)

    case "ramp" =>
      setUp(scn.inject(rampUsers(users) over (rampDuration)).
            protocols(httpConf)).maxDuration(duration)

    case unknown => println("Unknown scenario type : "+unknown)
  }
}
```

Izsek 32: Izvedba Gatling simulacije (stabilen in naraščajoč scenarij)

Gattling simulacijo je mogoče upravljati iz ukazne lupine. Najpomembnejša ukaza sta:

- $> sbt test: ukaz zažene obremenitveni test, upoštevajoč nastavitve
- $> sbt lastReport: ukaz prikaže rezultat zadnjega testa v privzetem brskalniku

Rezultati testiranja

ChessCloud storitev je nastavljena tako, da vsako izvajalsko vozlišče doda v gručo 3 izvajalce, kar pomeni, da je sistem sposoben analizirati 12 hkratnih zahtev (po tri na vseh štirih izvajalskih vozliščih). Analiza je omejena časovno na 5 sekund (storitev omogoča tudi omejitev analize glede na globino preiskanega prostora), zaradi česar je teoretični maksimum storitve 2,4 analize na sekundooziroma 148 analiz na minuto.

Spodaj so navedeni rezultati različnih scenarijev, ki kažejo, da sistem deluje pravilno tudi pod obremenitvami in da je kapaciteta oziroma zmogljivost sistema blizu teoretičnega maksimuma. S pomočjo stabilnih scenarijev je bil izračunana porazdelitev časa odgovora

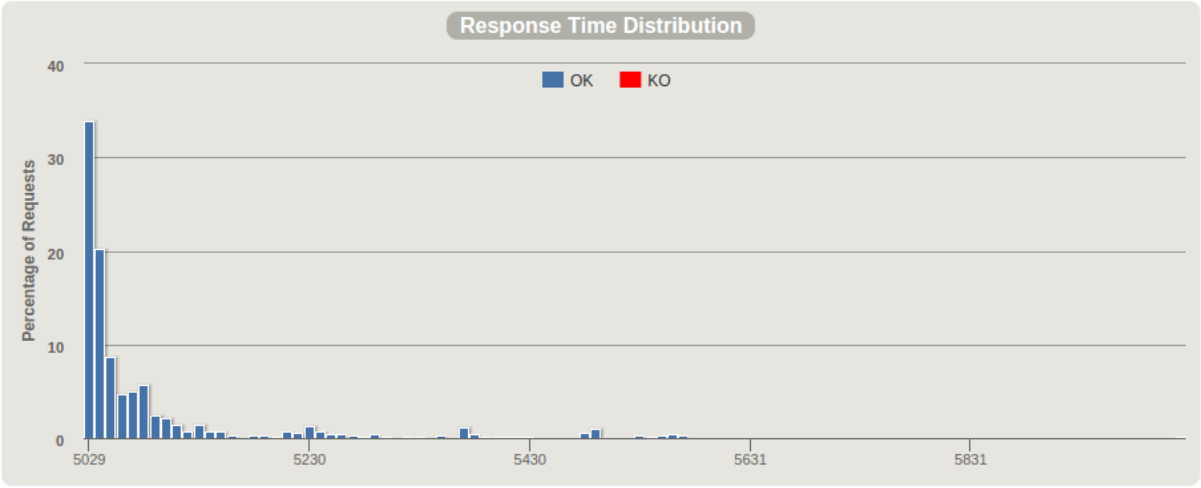
(angl. response time), s pomočjo naraščajočih scenarijev pa je nakazana kapaciteta sistema in njegovega obnašanja, ko se število zahtevkov viša nad to mejo.

Scenarij		**Izvedba**		**Odzvini časi (ms)**			
Uporabniki	Čas	Zahteve/s	Vseh/Napak	Min	Max	Povprečje	Std. dev
8	5 min	1,94	467/0	5026	6239	5078	119
10	5 min	1,94	585/0	5054	5721	5088	126
12	5 min	2,33	698/0	5024	6027	5085	111
14	5 min	2,37	707/0	5035	10111	5883	1173
16	5 min	2,33	700/0	5040	10702	6748	1433
18	5 min	2,36	708/0	5040	10255	7500	1759
20	5 min	2,36	708/0	5043	10975	8354	1909

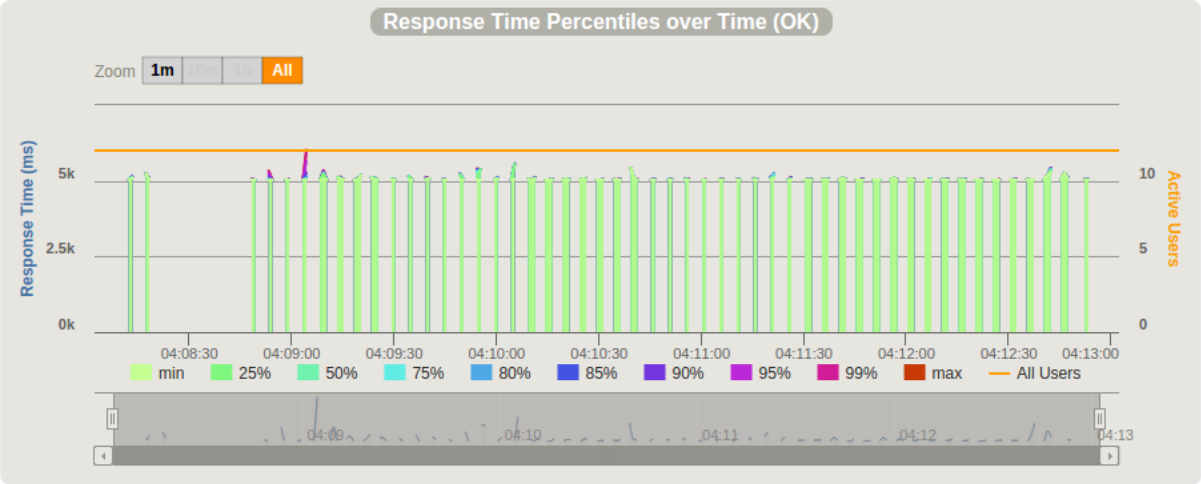
Tabela 3: Rezultati stabilnega 5-minutnega scenarija (število n.u. varira)

Tabela prikazuje razultate testiranj s stabilnim scenarijem dolžine 5 minut z različnim številom navideznih uporabnikov. Kot je razbrati iz zadnjih dveh kolon (poprečje in standardna deviacija oz. odklon), so odvizni časi za scenarije do 12 virtualnih uporabnikov podobni in enako razpršeni. Za scenarije, kjer je navideznih uporabnikov več kot je izvajalcev, pa se poprečni odzivni časi večajo sorazmerno s številom uporabnikov.

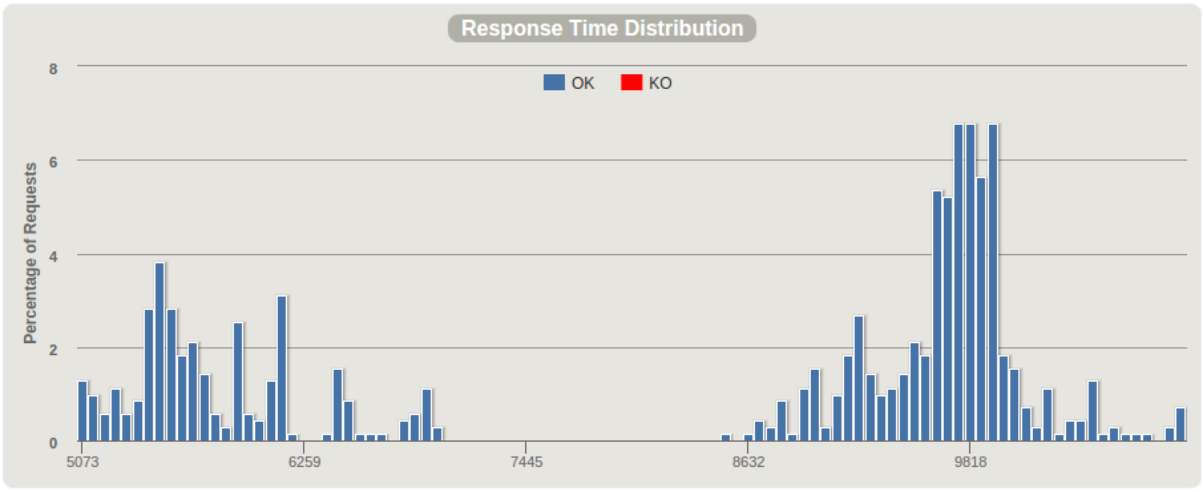
Spodaj so primeri vizualizacije, ki prikazujejo razpršenost odzivnih časov za 5-minutni scenarij za 12 in 20 navideznih uporabnikov. Prvi dve vizualizaciji prikazujeta, da je večina odzivov v območju petih sekund, medtem ko se pri drugem scenariju oblikujeta dve skupini. Prva okoli 5 sekund in druga okoli 10 sekund, kar nakazuje, da je nekaj zahtev serviranih takoj, ostale pa čakajo nekaj sekund v vrsti na prostega



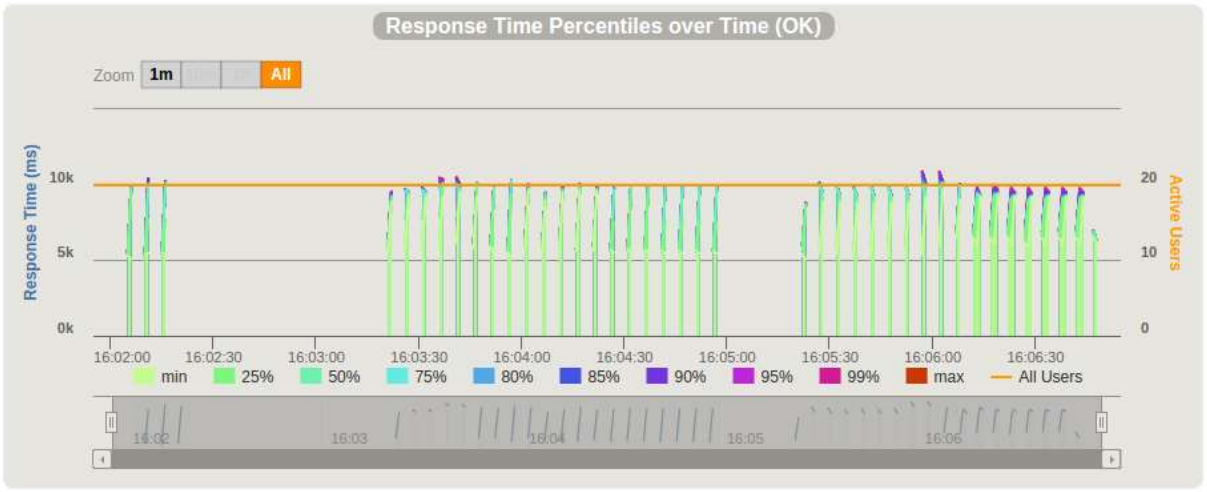
Slika 25: Razpršenost odzivnih časov (stabilni scenarij, 5-min, 12 n.u.)



Slika 26: Percentili odzivnih časov (stabilni scenarij, 5-min, 12 n.u.)

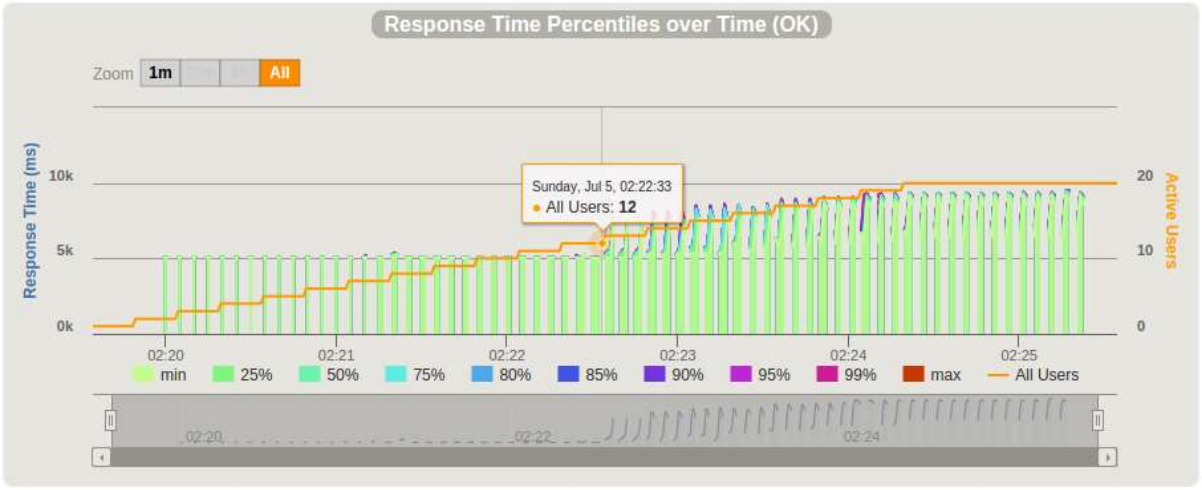


Slika 27: Razpršenost odzivnih časov (stabilni scenarij, 5-min, 20 n.u.)

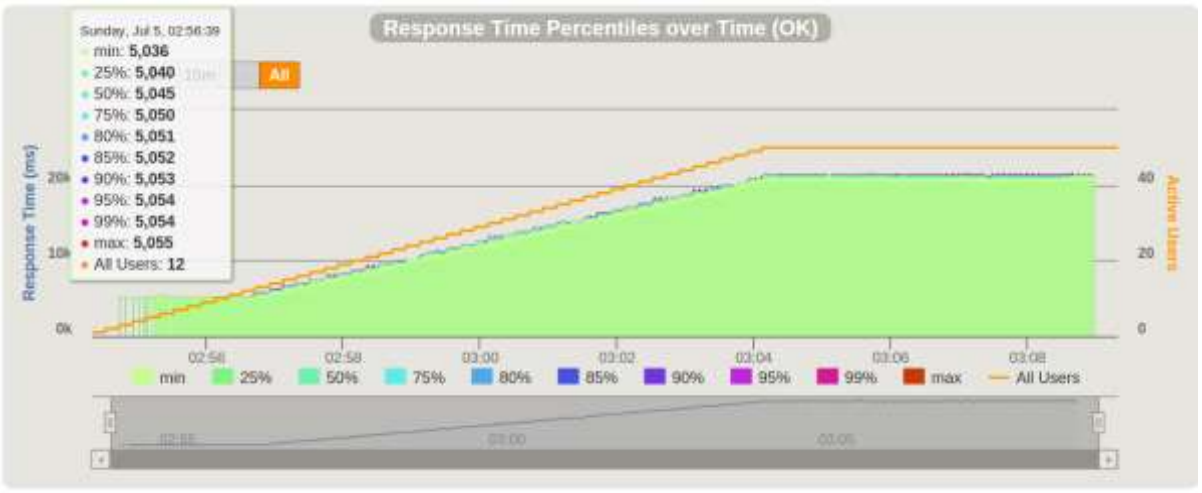


Slika 28: Percentili odzivnih časov (stabilni scenarij, 5-min, 20 n.u.)

Za podkrpitev rezultatov sta bila testirana dva naraščajoča scenarija, ki v prvem delu linearno povečata virtualne uporabnike do željene vrednosti, nato pa držita enako obremenitev še za določen čas. Dobljeni razultati kažejo, da odzvini časi do 12 virtualnih uporabnikov ostajajajo konstanti ali nespremenjeni, nato pa se začenjo linearno podaljševati, dokler število virtualnih uporabnikov raste. V zadnjem delu scenarija, ko je doseženo maksimalno oziroma najvišje mogoče število navideznih uporabnikov, ostajajo časi spet konstanti. Spodaj sta prikazani vizualizaciji ali vidni predstavitvi, ki nazorno prikažeta rezultate.



Slika 29: Odzivni časi (naraščajoči scenarij, 6-min, 20 n.u. v 5-min)



Slika 30: Odzivni časi (naraščajoči scenarij, 15-min, 50 n.u. v 10-min)

Ker so vsi konstantni oz. stalni testi dolžine 5 min, je spodaj še dodatni set scenarijev, ki primerja storitev pri uporabi istega števila virtualnih ali navideznih uporabnikov pri različnih dolžinah obremenitve.

Scenarij		Izvedba		Odzvini časi (ms)			
Uporabniki	Čas	Zahteve/s	Vseh/Napak	Min	Max	Poprečje	Std. dev
12	1	2,2	132/0	5026	6126	5151	279
12	5	2,33	698/0	5024	6027	5085	111
12	10	2,35	1408/0	5026	7089	5087	149
12	15	2,36	2123/0	5026	5890	5064	80

Tabela 4: Rezultati stabilnega scenarija z 12 n.u. (dolžina scenarija varira)

Tabela vsebuje test stabilnega scenarija z dvanajstimi navideznimi uporabniki pri različnih dolžinah scenarija (1 minuta, 5 minut, 10 minut in 15 minut). Iz rezultatov je razvidno, da poprečja in standardni odkloni ali deviacije ostajajo enake, kar nakazuje, da sistem deluje stabilno. Večje odstopanje je moč zaslediti samo pri minutnem scenariju, kar je povezano s tem, da so začetni odzivni časi malo večji, ker vse zahteve pridejo hkrati, kasneje pa se zaradi različnih odzivnih časov razpršijo.

Poglavje 7 Sklepne ugotovitve

V diplomskem delu je bila razvita skalabilna oblačna storitev za analiziranje šahovksih pozicij, imenovana ChessCloud. Uporabniku preko enostranske spletne aplikacije ponudi dostop do enega najmočnješih šahovskih strojev in omogoči analize šahovskih partij kot tudi neposredno igranje s šahovskim strojem. Storitev je nameščena v zasebnem oblaku OpenStack na zmogljivi infrastrukturi, kar omogoča kakovostne analize šahovskih pozicij, obenem pa zaradi storitvene narave razbremeni lokalne vire, ki so omejeni predvsem na mobilnih napravah. Storitev ChessCloud je javno dostopna na naslovu http://www.chess-cloud.com.

Pri realizaciji storitve diplomsko delo poleg samega razvoja zajema tudi nameščanje storitve v oblak, upravljanje z njo ter testiranje. Ker postopki in uporabljene tehnologije niso vezane na domeno računalniškega šaha, se lahko diplomsko delo uporabi tudi kot pomoč za prenos ostalih računsko intezivnih namiznih aplikacij v skalabilno oblačno okolje.

Storitev je razdeljena na dva dela. Sestavljata ga analitična gurča in spletna aplikacija, kjer je prvi del namenjen izvajanju analize poljubnih šahovskih pozicij, drugi del pa izpostavi to funkcionalnost uporabniku.

Analitična gruča omogoči analizo poljubnih šahovskih pozicij in je razvita na podlagi vzorca *delivec-izvajalec* po principu *pusti-da-pade*, ki sta se izkazala za zelo primerna za takšne vrste storitev, saj je gurča odporna na napake v posameznih vozliščih. Razviti so bili trije tipi vozlišč (*delivec, izvajalec in proizvajalec*), ki skupaj z natančno določenimi pravili komunikacije omogoča horizontalno skaliranje vsakega izmed tipov, kar zagotavlja lahko prilagajanje gruče obremenitvam spletne storitve. Analizo šahovske pozicije izvede šahovksi stroj Stockfish na izvajalskem vozlišču, katerega nadzoruje *izvajalec.* Ta s pomočjo UCI vmesnika nastavlja stroj in sporoča zahtevke za analizo, njegov izhod pa razčlenjuje in rezultate analize pošilja nazaj uporabniku.

Spletna aplikacija je razvita kot odzivna (angl. responsive) enostranska aplikacija (angl. single page application), ki se lahko s pomočjo različnih tehnologij (WebSocket, Comet, GET) sporazumeva s strežniškim delom, ki je zadolžen za posredovanje zahtevkov za analizo analitični gurči. Aplikacija omogoča pregled in analizo šahovskih partij kot tudi igranje s

šahovskim strojem. Poudarek pri razvoju aplikacije je bil na prijaznosti uporabe; prikazovnje vmesnih rezultatov analize, pregled analiziranih poti, upravljanje s tipkovnico, prilagodljivost na manjših zaslonih itd.

Za namene postavitve storitve na oblačno infrastrukturo so bile razvite skripte, ki preko ukazne lupine omogočajo izvajanje ukazov, potrebnih za pripravo infastrukture, horizontalno skaliranje vozlišč, pripravo paketov storitev in njihovo nameščanje, upravljanje s storitvijo itd. Pri samem razvoju in razhroščevanju storitve so bile te skripte v izredno pomoč, saj so avtomatizirale sicer dolgotrajajoče postopke. Skripte podpirajo nameščanje na dva oblačna ponudnika; zasebni OpenStack in javni Amazon AWS EC2 oblak. Storitev se namešča na Debian Linux in je zapakirana v DEB pakete, ki ob namestivi priravijo upstart servis ter novega uporabnika, z omejenimi pravicami, ki se uporablja pri zagonu servisa. Izvajajočo storitev je mogoče nadzorovati, poleg s sistemom, ki ga ponudi ponudnik oblaka (npr. Ganglia na OpenStacku), s pomočjo NewRelic komercialno rešitvijo, ki v zastonjski različici ponuja napreden nadzor nad infrasturkturo in aplikacijo. Slednja se je izkazala za koristno, saj sporoča kritična stanja vozlišč in stanje analitične gruče, ko zaradi preobremenjenosti zasebna postavitev OpenStacka začne delovati nepredvidljivo, kar lahko povzroči razpad gurče.

Za preverjanje pravilnosti delovanja sta bila razvita preprosta Gattling scenarija, ki simulirata navidezne uporabnike pri zahtevanju analiz šahovske pozicije. Prvi scenarij ima stabilno bazo uporabnikov, drugi pa naraščajočo. S scenarijema je bilo moč preveriti, ali se storitev oziroma analitična gurča obnaša pričakovano in stabilno tudi pri povečani obremenitvi.

S pomočjo raziskovanja računalniškega šaha in oblačnih okolij, opisanih v teoretičnem delu diplomske naloge, in z vključevanjem modernih tehnologij, ki so dandanes nujno potrebne pri razvoju skalabilnih oblačnih sistemov, je bila realizirana skalabilna oblačna storitev za analiziranje šahovskih pozicij, ki deluje zanesljivo in predstavlja funkcionalno (primerno, dobro) zaokroženo celoto. Možnosti za izboljšave in nadaljnje delo pa je seveda ogromno, ampak kot vsako stvar je bilo potrebno tudi to delo nekje zaključiti.

Seznam slik

Seznam izsekov

Seznam tabel

Literatura

[1] D. Shenk, "The Immortal Game: A History of Chess": Anchor, 2007.

[2] E. Gamma, R. Helm, R. Johnson in J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software": Addison-Wesley, 1995

[3] S. J. Edwards, "Standard: Portable Game Notation Specification and Implementation Guide" [Online]. Dosegljivo: http://www.saremba.de/chessgml/standards/pgn/pgn-complete.htm

[4] FIDE LAWS of CHESS, Handbook [Online]. Dosegljivo: http://www.fide.com/FIDE/handbook/LawsOfChess.pdf

[5] Description of the universal chess interface (UCI), Specification [Online]. Dosegljivo: https://github.com/clojure-dus/chess/blob/master/uci-specification.txt

[6] Chess Engine Communication Protocol (CECP), Specfiicaiton [Online]. Dosegljivo: http://www.gnu.org/software/xboard/engine-intf.html

[7] CEGT (Chess Engines Grand Tournament), Website [Online]. Dosegljivo: http://www.husvankempen.de/nunn/

[8] CCRL (Computer Chess Rating Lists), Website [Online]. Dosegljivo: http://www.computerchess.org.uk/ccrl/"

[9] P. Mell in T. Grance, "The NIST Definition of Cloud Computing", 2011 [Online]. Dosegljivo: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[10] T. Höllwarth, "Cloud Migration": Verlagsgruppe Hüthig Jehle Rehm, 2012.

[11] Cloud Computing, Wikipedia [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Cloud_computing

[12] Gossip protocol, Wikipedia [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Gossip_protocol

[13] Amazon Web Services, Website [Online]. Dosegljivo: http://aws.amazon.com/

[14] OpenStack, Website [Online]. Dosegljivo: https://www.openstack.org/

[15] Apache LibCloud, Website [Online]. Dosegljivo: https://libcloud.apache.org/

[16] Fabric, Website [Online]. Dosegljivo: http://www.fabfile.org

[17] Paramiko, Website [Online]. Dosegljivo: http://www.paramiko.org/

[18] Playframework, Website [Online]. Dosegljivo: https://www.playframework.com/

[19] Akka, Website [Online]. Dosegljivo: http://akka.io/

[20] Nginx, Website [Online]. Dosegljivo: http://www.nginx.org/

[21] AngularJS, Website [Online]. Dosegljivo: https://www.angularjs.org

[22] ChessBoardJS, Website [Online]. Dosegljivo: http://chessboardjs.com/

[23] ChessJS, Website [Online]. Dosegljivo: https://github.com/jhlywa/chess.js

[24] Ganglia, Website [Online]. Dosegljivo: http://ganglia.sourceforge.net

[25] Gatling, Website [Online]. Dosegljivo: http://www.gatling.io/